Design Patterns com PHP 7

Desenvolva com as melhores soluções

Casa do Código

GABRIEL ANHAIA

Sumário

- ISBN
- Partilhe este livro
- Agradecimentos
- Sobre o autor
- Prefácio
- Início
  - 1 Padrões de Projeto no PHP 7.2
- Padrões de criação
  - 2 Um ponto global de acesso com Singleton
  - 3 Construindo objetos com Factory Method
  - 4 Prototipação de objetos com Prototype
  - 5 Objetos complexos com Builder
- Padrões estruturais
  - 6 Tudo se encaixa com Adapter
  - 7 Descomplicando o sistema com Facade
  - 8 Agregando funcionalidades com Decorator
- Padrões comportamentais
  - 9 Capturando o estado atual de um objeto com Memento
  - 10 Atualização em tempo real com Observer
  - 11 Variando partes de um algoritmo com Template Method
  - 12 A mudança de comportamentos em tempo de execução com Strategy
- Conclusão
  - 13 Dicas de leitura
  - 14 E agora?
  - 15 Referências bibliográficas

ISBN

Impresso e PDF: 978-85-94188-63-2

EPUB: 978-85-94188-64-9

MOBI: 978-85-94188-65-6

Caso você deseje submeter alguma errata ou sugestão, acesse http://erratas.casadocodigo.com.br.

Partilhe este livro

Gosta de ler? Então compartilhe suas leituras, recomende os livros que gostou de ler, comente sobre resenhas, partilhe!

Confio em vocês! partilhem!

Agradecimentos

É capaz quem pensa que é capaz. – Buda

Com esta frase, comecei este novo projeto, correndo atrás de mais um sonho!

Muitas vezes, torna-se difícil planejar como chegar a algum objetivo, dar o primeiro passo, e seguir diariamente motivado a buscar algo mais. Entretanto, posso dizer que felizmente não estou nessa caminhada sozinho, e só tenho a agradecer por todos que até agora compartilharam comigo esta grande jornada chamada vida. Agradeço a todas as pessoas que conheci, que ajudaram de alguma forma a moldar minha personalidade, seja por uma ação, por uma palavra, ou simplesmente por estarem lá.

Sou muito grato por minha mãe **Vera Regina**, que sempre me mostrou o quanto uma pessoa pode ser bondosa com os outros, ajudando o próximo. Ela, sem dúvida, é uma das pessoas mais honestas que conheço, e disto me orgulho muito. Ao meu pai **Paulo Roberto**, em quem sempre vi um lado empreendedor, disposto a montar qualquer negócio e quem sempre me ajudou. Também à minha segunda mãe, **Ana Luiza**, que praticamente me criou, e à minha vó **Geraldina Anhaia**, que não poderia me esquecer de citar. Muito obrigado por tudo, sou muito feliz em ser seu neto!

Também sou grato a todos os meus familiares e amigos, meus colegas de trabalho com quem sempre pude compartilhar conteúdos relacionados a tecnologia. Todos foram fundamentais em minha caminhada até aqui.

Um agradecimento especial a meu amigo de infância e CEO da Goal Construções, **Braian Ottoni**, que me deu uma visão de quanto o esforço pode nos levar aonde queremos, e por me ouvir milhares de vezes falar sobre o desenvolvimento deste livro. Ao meu colega **Anderson Scherer**, especialista em tecnologias **front-end**, na qual

desenvolvemos inúmeros projetos desafiadores feitos sob medida para várias empresas.

Agradeço também ao meu primo **Lucas Azevedo**, criador do projeto *Um livro para um desconhecido*, que tem o propósito de compartilhar conhecimento através de um gesto simples, a doação de um livro para uma pessoa desconhecida.

Agradeço a meu ex-colega, amigo e um excelente programador **Adalberto da Silva Maria**, que me serviu como um dos revisores da parte técnica deste conteúdo, e fez um trabalho fantástico revisando-o em um curto prazo de tempo, mas, mesmo assim, retornando vários detalhes para ajustar.

Por fim e principalmente, agradeço à **Vivian Matsui** e à **Bianca Hubert**, que me ajudaram muito a estruturar este livro e fizeram um excelente trabalho de revisão. Só tenho a agradecer à **Casa do Código** e ao grupo **Caelum** por esta oportunidade!

Sobre o autor

Figura -1.1: Gabriel Anhaia

Meu nome é Gabriel Anhaia, sou formado em Análise e Desenvolvimento de Sistemas e trabalho com desenvolvimento de

software profissionalmente há mais de 6 anos. Na maior parte desse tempo, estou focado na linguagem de programação PHP, embora já tenha desenvolvido algumas coisas em Java, NodeJS e algumas aplicações mobile.

Sou fanático por empreendedorismo, o mundo dos negócios e suas possibilidades. Hoje, diria que sou mais um empreendedor do que apenas um desenvolvedor.

Trabalho como *freelancer* em uma fábrica de *software*, e também possuo um blog (www.mestredev.com.br), no qual publico constantemente conteúdos relacionados à área da tecnologia e empreendedorismo.

No início de minha carreira, trabalhei alguns anos com eletrônica na indústria. Depois disso, trabalhei mais dois anos na equipe de aviação do Aeroporto Internacional Salgado Filho (Porto Alegre/RS), fazendo a manutenção eletrônica de aeronaves, até o momento que decidi migrar para a área de desenvolvimento de software

Desde então, trabalhei com os mais variados tipos de sistemas, desenvolvi aplicações em cima de diferentes *frameworks*, tanto de projetos novos (projetados desde as etapas de análise) até de sistemas legados que já eram usados há vários anos. Tive contato com muitos tipos de arquiteturas, o que me fez observar o quão bom ou ruim é a forma como uma aplicação pode ser construída.

Acredito que o desenvolvimento de um sistema deve ser feito sempre buscando a excelência e a aplicação das boas práticas – independente de linguagem –, sempre buscando a máxima qualidade!

- LinkedIn: https://www.linkedin.com/in/gabrielanhaia

Prefácio

Este livro mostra a fundo o uso dos mais variados tipos de Padrões de Projeto (*Design Patterns*), da forma mais objetiva e clara possível. São apresentadas as melhores soluções para problemas específicos encontrados constantemente na vida de um desenvolvedor de *software*. Não é um guia para Orientação a Objetos, porém faz uso dos conceitos mais básicos deste paradigma.

Os Padrões de Projeto são independentes de qualquer linguagem, e, no geral, não são complexos depois que entendemos sua ideia. Com toda certeza, posso afirmar que, depois de aprender Padrões de Projeto, sua visão sobre sistemas orientados a objetos muda completamente.

Com eles, conseguimos desenvolver sistemas mais modulares, expansíveis, reutilizáveis e com mais flexibilidade. Podem até ser combinados para se tirar um proveito ainda maior. Eles também são considerados como uma *skill* fundamental para qualquer desenvolvedor da atualidade.

A quem se destina este livro?

Este livro é destinado a desenvolvedores de software que já tiveram contato ou têm algum conhecimento em Orientação a Objetos e a linguagem de programação PHP. Ele foi feito principalmente para quem quer aprender a implementar Padrões de Projeto em suas aplicações e, no geral, para quem quer aumentar a qualidade dos seus projetos a um outro nível.

Considera-se que o leitor já possua o conhecimento básico necessário para executar scripts e aplicações PHP em um servidor-web de sua preferência (independente do sistema operacional). Também pode ser usado como referencial teórico para outras linguagens de programação além do PHP, por se tratar de um tema mais genérico.

Como devo estudar?

No decorrer do livro, serão apresentados os mais variados Padrões de Projeto, sendo que cada padrão possui um tipo e um nível de dificuldade de implementação. Todas essas informações serão especificadas. Ao passar por cada um deles, serão apresentados descrições, problemas reais que ele se propõe a resolver, diagramas, dicas e, principalmente, uma parte prática a ser exercitada pelo leitor. Essa última parte foi elaborada para que as aplicações reais dos padrões fiquem o mais claro possível.

Ferramentas utilizadas

Nossos exemplos de implementação de código foram desenvolvidos a partir da versão 7.2 do PHP, porém, a partir da versão 7.0, já é possível executá-los. A IDE usada para escrever os códigos foi a **PHP Storm 2018**. Entretanto, qualquer editor de texto já é o suficiente para executar os testes (Sublime, Notepad++, Visual Studio Code etc.). Como servidor web para a execução dos scripts, utilizei o Apache, instalado em um Ubuntu 16.

Exemplos de código

Todos os exemplos de código descritos neste livro podem ser encontrados em um repositório que criei em meu GitHub: https://github.com/gabrielanhaia/design-patterns-php7.

Uma observação é que estou utilizando *namespaces* e um *autoloader* simples para o carregamento automático das classes. Desta forma, não é necessário ficar carregando todas manualmente com o uso de funções como o `include` ou `require`.

Para usá-los, basta copiar o arquivo `autoloader.php` dentro do repositório e seguir a mesma hierarquia de pastas, conforme especificado nos capítulos seguintes do livro.

Desejo a você uma ótima leitura!

Início

Ao longo desta parte inicial, serão abordados conceitos importantes de Programação Orientada a Objetos, boas práticas de desenvolvimento e algumas das principais novidades do PHP 7.2.

Capítulo 1
Padrões de Projeto no PHP 7.2

1.1 Os Padrões de Projeto de software

Definição

Do inglês *Design Patterns*, podemos definir Padrões de Projeto como modelos de soluções para algum problema específico encontrado frequentemente dentro de um projeto de *software*. Eles servem como *templates* a serem aplicados para desenvolver uma solução para os problemas.

Não são códigos implementados que podem ser copiados para outros softwares (na maioria das vezes), mas apenas a definição de sua aplicação. Por isso, eles funcionam nos mais variados tipos de escopos e para diversos fins. Você pode aplicar um Padrão de Projeto específico tanto em um jogo quanto em um site de vendas, tudo depende de sua necessidade.

Outra consideração é que Padrões de Projeto são independentes da linguagem usada. Eles funcionam na maioria das linguagens orientadas a objetos, no entanto, podem conter algumas diferenças em suas implementações – o que vai depender das funcionalidades disponíveis e das peculiaridades de cada uma dessas linguagens.

Quando aplicados de maneira correta e para uma necessidade real, os Padrões de Projeto são considerados ótimas práticas de desenvolvimento.

A origem dos Padrões de Projeto

Tudo começou em 1987, quando Kent Beck e Ward Cunningham abordaram os primeiros Padrões de Projeto de software em uma conferência nos Estados Unidos, desenvolvidos em *Smalltalk* (linguagem popular na época). Porém, os padrões começaram a ganhar forma apenas em 1995 a partir da publicação do livro *Design Patterns: Elements of Reusable Object-Oriented Software*, escrito por Erich Gamma, Richard Helm, Ralph Johnson e John Vlissides, que passaram a ser conhecidos como a "Gangue dos Quatro" (ou GOF).

Existem muitos Padrões de Projeto documentados na *web*, mas, sem dúvida, os desenvolvidos pelo **GOF** são os mais conhecidos. Ao total, são 23 padrões divididos em três categorias:

1. Padrões de criação;
2. Padrões estruturais;
3. Padrões comportamentais.

Padrões de criação

Responsáveis por abstrair a criação de objetos, eles transferem e centralizam responsabilidades de como e onde instanciá-los, tornando o sistema mais flexível. Os padrões de criação definidos pelo *GOF* são:

- Abstract Factory;
- Builder;
- Factory Method;
- Prototype;
- Singleton.

Padrões estruturais

Os padrões estruturais definem como classes e objetos são compostos entre si, permitindo alterações nessa composição em

tempo de execução. Estão mais relacionados à estrutura como as classes e objetos estão ligados.

Os padrões estruturais definidos pelo *GOF* são:

- Adapter;
- Bridge;
- Composite;
- Decorator;
- Facade;
- Flyweight;
- Proxy.

Padrões comportamentais

Eles definem a comunicação entre classes e objetos, como também de seus comportamentos. Estão ligados diretamente à atribuição de responsabilidades desses objetos e como os algoritmos se comportam.

Os padrões comportamentais definidos pelo *GOF* são:

- Chain of Responsibility;
- Command;
- Interpreter;
- Iterator;
- Mediator;
- Memento;
- Observer;
- State;
- Strategy;
- Template Method;
- Visitor.

1.2 Orientação a Objetos

Conceitos

Também conhecida pela sigla **POO** (paradigma orientado a objetos), trata-se de um paradigma de programação criado para resolver alguns problemas da **Programação Procedural**. Com a Orientação a Objetos (OO), é possível termos um maior aproveitamento de código, uma melhor organização de nossos projetos e uma codificação comparável a cenários do mundo real (com uma melhor compreensão).

A maioria das linguagens de programação populares atualmente segue esse paradigma, justamente por todas essas vantagens trazidas por ela. Aliás, uma delas foi a criação dos **Padrões de Projeto de Software**.

Além desse paradigma, também existem outras opções como **Programação Funcional**, **Programação Procedural**, entre outras. Não necessariamente uma é melhor que a outra, pois tudo depende da situação e do problema que queremos resolver.

Associação, agregação e composição

Quando trabalhamos com o paradigma orientado a objetos, é importante conhecer a forma como os objetos podem se relacionar entre si: associação, agregação e composição. A *associação* pode ser definida quando possuímos um objeto que contém uma referência de outro, porém ambos são independentes; um não precisa do outro para existir.

Um exemplo disso seriam vários objetos do tipo `Aluno` estarem associados a um objeto do tipo `Professor`, e um único objeto `Aluno` estar associado a vários objetos de `Professor`. Entretanto, ambos podem existir sozinhos.

A *agregação* é semelhante à associação, mas temos um objeto que faz uso de outros. Ele tem uma dependência para executar suas tarefas. Um exemplo disso é uma classe `CarrinhoDeCompras` que

precisa de instâncias da classe `Produto` para executar suas operações, sendo que os produtos estão agregados ao carrinho.

A composição é quando um objeto possui instâncias de outros inseridas ao objeto principal em sua construção. Quando o objeto principal é destruído, os objetos que o compõem serão destruídos juntos.

Princípios de SOLID

SOLID é um acrônimo de alguns princípios da Programação Orientada a Objetos, citados pelo famoso engenheiro de software **Robert C. Martin**. A seguir, veja cada um deles:

Letra	Significado	Definição
S	Princípio de responsabilidade única	Uma classe deve ter apenas um motivo para mudar.
O	Princípio de aberto-fechado	Deve ser possível estender o comportamento de uma classe, sem modificá-lo.
L	Princípio da substituição de Liskov	As classes-base devem ser substituíveis por suas derivadas.
I	Princípio da segregação da interface	É melhor possuir muitas interfaces específicas do que uma única.
D	Princípio de inversão de dependência	Depender de abstrações, e não de implementações.

Assim como os Padrões de Projeto, os princípios são independentes de linguagem ou tecnologia. São princípios a serem seguidos sempre que necessário para se tirar o máximo proveito da OO.

Com a aplicação deles, temos diversas vantagens que a Orientação a Objetos nos proporciona. Entre eles, temos uma maior facilidade na manutenção e expansão de um software, os algoritmos tornam-se mais testáveis na maioria das vezes e temos um maior aproveitamento de código.

1.3 PHP 7.2

Foi a partir do ano de 2015 que o PHP começou a tomar um novo rumo, quando foi lançada a versão 7. Em 2016, foi lançada a versão 7.1 e, logo após, a 7.2 (atual), que trouxeram diversos recursos. Acredito que a mais impactante foi em relação à sua performance, através de testes de *benchmark*. Obteve-se resultados de velocidade na execução do código até 9 vezes mais rápido, comparado a suas versões anteriores.

Entretanto, não foi só a performance que melhorou. Entre outras alterações, muitas funções foram removidas ou depreciadas. Também houve uma alteração nos construtores das classes: até a versão anterior, era possível usar o método `__construct`, ou criar um método com o próprio nome da classe; essa segunda opção foi removida.

Não podemos deixar de citar três novidades em particular. A primeira é o funcionamento dos erros fatais e exceções. Diferente das versões anteriores, em que ocorriam erros fatais gerados dentro da aplicação – parando nosso programa –, agora o próprio PHP lança as exceções. Desta forma, podemos tratá-las através das cláusulas `try` e `catch`, tudo isso em tempo de execução e sem parar a execução da aplicação.

A outra grande alteração foi a introdução dos *Scalar Types*. Como sabemos, o PHP é uma linguagem de programação "não tipada", porém, com a adição deste recurso, podemos considerá-lo uma

linguagem "fracamente tipada". Agora, é possível definir nos parâmetros de métodos o tipo de dado obrigatório que ele recebe. Este pode ser `int`, `float`, `string`, `bool`, `array` ou até o nome de uma classe criada por nós.

Assim, é possível construir aplicações menos suscetíveis a erros de desenvolvimento. Anteriormente, não havia controle nenhum sobre os tipos de dados. Segue um exemplo de uso dos *Scarlar Types*:

```
public function metodoTeste(int $numeroInteiro, string $texto, MinhaClasse $minhaClasse)
{
    ...
}
```

A mudança final é a definição de tipo de retorno de métodos e funções. Agora, é possível definir o tipo de dado que uma função ou método vai retornar obrigatoriamente, gerando um erro caso isso não ocorra. Segue o exemplo da definição de tipo de retorno:

```
public funtion metodoTesteRetorno(): string
{
    return 'teste';
}
```

PHP orientado a objetos

Em aplicações em PHP, nem sempre foi possível o desenvolvimento orientado a objetos. Quando a linguagem foi criada em 1994, os códigos eram feitos de forma procedural, orientados a funções. Só a partir da versão 3, o PHP começou a sofrer alterações significativas para este lado, e, após o lançamento da versão 5.3, estava consolidado como uma linguagem que suportava o paradigma da OO.

O PHP começou a suportar o uso de estruturas de classes, interfaces, entre outras coisas. Não era mais necessário o desenvolvimento orientado a funções. Hoje, a linguagem conta com uma infinidade de recursos comparável à maioria das linguagens de

programação de alto nível, e está entre as mais utilizadas do mundo.

1.4 Boas práticas de programação com PHP

Desenvolver um código seguindo boas práticas de programação só traz vantagens. Podemos obter uma maior organização de nosso código, melhorar o processo de desenvolvimento, facilitar a expansão, a manutenção e a redução dos custos de desenvolvimento. Com isso, temos o aumento da qualidade final. Além da aplicação de Padrões de Projeto, que é o princípio deste livro, a seguir fica a sugestão de algumas boas práticas a serem seguidas no desenvolvimento de seu código.

Nomes significativos

Sempre que for nomear algo em sua aplicação, uma boa prática é colocar um nome que faça sentido para o que se quer nomear. Pergunte-se: "O que isso representa?", "O que isso vai fazer?" etc.

Um exemplo disso são as variáveis `$x` ou `$cont`. Temos de pensar que estas foram criadas para um propósito, como armazenar algum valor específico, seja ele um contador ou qualquer outra coisa. Por que não substituir `$cont` por `$contador`? Em uma futura manutenção, com certeza a segunda opção seria compreendida muito mais facilmente por um desenvolvedor. Isso vale também para o nome de classes, interfaces e qualquer outra coisa que possa ser nomeada.

Comentários e documentação

Outra boa prática de desenvolvimento é a documentação de código, porém esta deve ser sempre feita com cuidado. Se um bloco de código está complexo demais e precisa ser explicado, devemos nos perguntar se o problema não é a complexidade desnecessária.

Jamais devemos sobrecarregar o código com comentários explicando o que cada linha faz. Os comentários e a documentação facilitam em muito o seu entendimento, mas quando aplicados de maneira moderada.

Outro grande benefício dos comentários e da documentação de código é a possibilidade de documentar os tipos de dados armazenados em propriedades quando utilizamos em conjunto com uma **IDE** (como o **PHP Storm**, por exemplo). A seguir, podemos ver um exemplo prático disso.

Ao declarar a classe:

```
class Calculadora
{
    public function calcular(){
        ...
    }
}
```

Agora, vamos utilizar a classe `Calculadora` como parâmetro de outra:

```
class Teste
{
    /** @var Calculadora $calculadora Objeto de calculadora do sistema **/
    public $calculadora;

    public function __construct()
    {
        $this->calculadora = new Calculadora();
    }
}
```

Como no PHP não é possível definir o tipo de dado armazenado em um atributo, estamos documentando a propriedade `$calculadora` e dizendo que ela é do tipo `Calculadora`. O benefício que temos ao utilizar essa documentação junto a uma **IDE** é o *autocomplete*, que nos sugere os métodos e as variáveis presentes dentro de `$calculadora`, como pode ser visto a seguir:

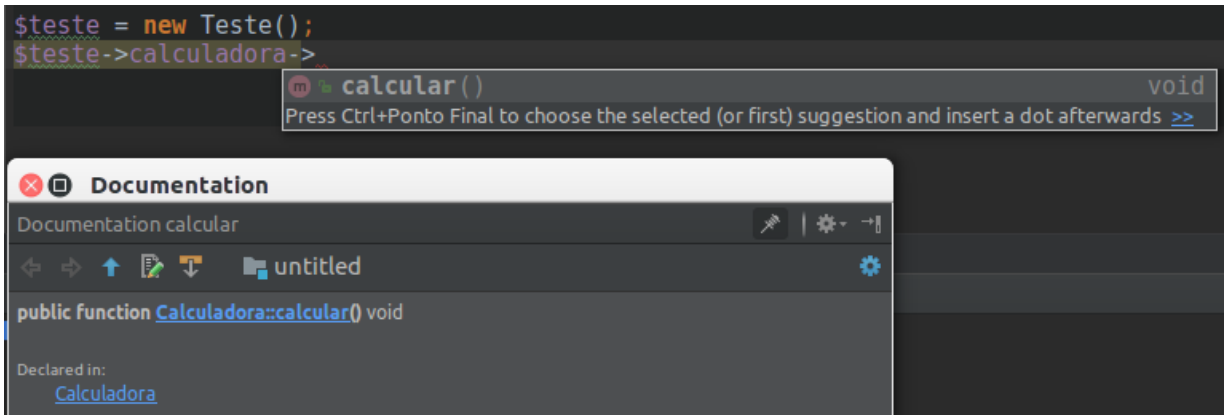

Figura 1.1: Teste de autocomplete na documentação de atributo de classe

Com isso, ganhamos maior velocidade e uma certa segurança no desenvolvimento de nossas aplicações.

Tratamentos de erros

Uma excelente prática – não só do PHP, mas de qualquer linguagem que suporte – é a utilização de `exceptions` e das cláusulas `try` e `catch`. Com seu uso, é possível tratar erros de execução de uma aplicação, sem que o software pare de funcionar. Com isso, podemos até segmentar nossos erros e ter uma organização melhor de nosso programa; basta criarmos classes que estendam da classe nativa `Exception` do PHP.

TDD e testes unitários

Práticas como o **TDD** e testes unitários vêm tornando-se cada vez mais comuns no desenvolvimento de software. O **TDD** (*Test-Driven Development*) é o desenvolvimento de software orientado a testes, que tem como ideia escrever testes do código antes mesmo de escrever o código. Quando escrevemos o teste antes de a parte a ser testada, garantimos que nosso sistema vai obrigatoriamente ser construído com o funcionamento esperado; caso contrário, os testes não vão passar.

Se compararmos com a ordem de desenvolvimento tradicional, na qual desenvolvemos o teste depois, muitas vezes acabamos construindo uma parte de nossa aplicação sem realmente saber como ela deve se comportar (seu funcionamento), para só depois desenvolvermos o teste. Muitas vezes, isso é sustentado por um desenvolvimento que não foi muito bem planejado, não houve grande esforço de análise do projeto, ou ele foi mal estruturado.

Outra prática muito usada, principalmente combinada ao TDD, é a adoção de **Testes Unitários**, que têm como objetivo testar comportamentos de partes isoladas de nosso sistema, chamadas de unidade. Com esses testes isolados, temos a possibilidade de uma maior cobertura de código. Eles incentivam a refatoração e a melhora da escrita, evitam a perda de tempo de *debug* na busca por erros, servem como documentação do que cada parte do código deve realmente fazer (quando bem escritos), entre muitas outras vantagens.

Atualmente, a principal ferramenta utilizada em PHP para a criação de testes unitários é a **PHP Unit**. Deixo como sugestão a leitura do livro *Test-Driven Development: Teste e Design no Mundo Real com PHP*, que pode ser encontrado em https://www.casadocodigo.com.br/products/livro-tdd-php/.

PSRs

As PSRs são especificações de projetos sugeridas e votadas por meio do grupo **PHP-FIG**. Essas especificações são voltadas para a codificação PHP de uma maneira geral, desde um simples espaçamento após um `if` até o padrão de uso de interfaces para *Logs*.

Vale salientar que nenhuma PSR é uma imposição; elas são tratadas apenas como sugestões. As PSRs são atualizadas constantemente e, para quem acompanha, sugiro ficar sempre atento ao site oficial, em https://www.php-fig.org/psr/.

O **PHP-FIG** (Framework Interop Group), iniciado em 2009, define diversos padrões de desenvolvimento com uso do PHP. Entre eles, podemos citar estrutura de diretórios, carregamento de classes e formatação do código. A ideia é que representantes de diferentes projetos debatam sobre a forma como desenvolvem e definem os padrões. O objetivo é aplicá-los em seus projetos, porém, na própria documentação, eles dizem que é totalmente aberto para todo e qualquer membro da comunidade PHP.

Entre os atuais membros responsáveis pelas decisões a respeito dos padrões, temos representantes dos principais nomes quando falamos de PHP: CakePHP, Drupal, PEAR, Phalcon, Symfony e Zend Framework 2.

Padrões de criação

Responsáveis por abstrair a criação de objetos, esses padrões transferem e centralizam as responsabilidades de como e onde instanciar estes objetos, tornando o sistema mais flexível.

Este livro aborda os seguintes padrões de criação:

- Singleton;
- Factory Method;
- Prototype;
- Builder.

Capítulo 2
Um ponto global de acesso com Singleton

Tipo: padrão de criação

Nível de dificuldade:

2.1 O problema dos logs descontrolados

Imagine que você administra um sistema de um banco de grande porte. Um banco precisa de segurança em suas transações e rastreabilidade de todos os possíveis erros de sistema que possam ocorrer. Nesse sistema, existe uma única classe responsável por gravar logs (registros para fins de histórico do que é feito no sistema).

A cada operação – ou seja, quase todas as ações executadas no sistema –, um log é gravado com diversas informações. Porém, à medida que o número de usuários aumentou, o sistema começou a apresentar problemas de performance. Todos os usuários começaram a reclamar constantemente da lentidão.

Um dos motivos para esse problema foi que, a cada vez que precisávamos gravar um log, tínhamos de instanciar a classe novamente; a cada nova instância, um espaço na memória do servidor era ocupado. Como o sistema recebia um número considerável de usuários simultâneos, isso acabou elevando o uso da memória física do sistema até seus limites.

Após chegarem à conclusão de que esse número elevado de instâncias de classe de log era um dos principais causadores do problema de performance, surgiu a necessidade de diminuir o número de instâncias simultâneas existentes. Se fosse possível instanciar esse objeto de logs uma única vez e compartilhá-lo todo o tempo de execução (período em que um programa de computador permanece em execução) do sistema, resolveríamos o problema; é ai que entra o padrão **Singleton**.

2.2 O padrão Singleton

Definição

Assim como o problema do sistema bancário apresentado anteriormente, no qual tivemos um problema de performance da aplicação por possuir um grande número de instâncias de uma classe – criadas simultaneamente em cada requisição ao servidor –, precisamos ter um maior controle da criação de instâncias descontroladas de uma classe.

O **Singleton** tem a finalidade de garantir esse controle, de gerenciar o número de instâncias criadas para uma determinada classe, e também centralizar a criação desses objetos em um único ponto de acesso global (que pode ser acessado de qualquer parte do sistema).

Existem diversos exemplos pela internet de implementações do **Singleton**, em que a classe principal controla a instância de outras classes. Porém, isso não garante o real controle de instâncias de uma classe, já que um desenvolvedor desavisado poderia ignorar a classe do *Singleton* (que controla a criação das instâncias) e criar novos objetos, seja por não ter conhecimento de sua existência ou não querer utilizá-la.

O correto é que a classe que segue esse Padrão de Projeto controle a criação de suas próprias instâncias, assim, não seria possível criar instâncias dela de outra forma. A forma de limitar a criação de novas instâncias da própria classe varia de linguagem para linguagem. Veremos todos os passos para fazer isso com o PHP 7+.

SINGLETON, O ANTI-PATTERN

Para muitos, o padrão **Singleton** é considerado um anti-pattern por conta do seu uso em forma estática (chamando a classe sem precisar instanciá-la) e, principalmente, pelo acesso global à instância da classe. Os problemas de utilizar classes estáticas são vários: não podemos trabalhar com interfaces, nosso código fica com um acoplamento mais alto, entre outros.

Também temos problemas relacionados a acesso global à variável da instância. Ele gera um difícil entendimento de por que a variável não está diretamente ligada a um escopo específico do sistema. Também temos o risco de uma variável que pode ser alterada em qualquer ponto do sistema acabar afetando outras partes.

Apesar de possuir todas estas desvantagens, a aplicação de qualquer padrão deve ser sempre medida de acordo com a real necessidade. No caso do **Singleton**, seu uso é indicado caso seja necessário o controle de acesso simultâneo a um recurso compartilhado, ou caso realmente precisemos de uma única instância.

Diagrama de classes

A seguir, veja o diagrama de classes do padrão **Singleton**, de acordo com o nosso exemplo:

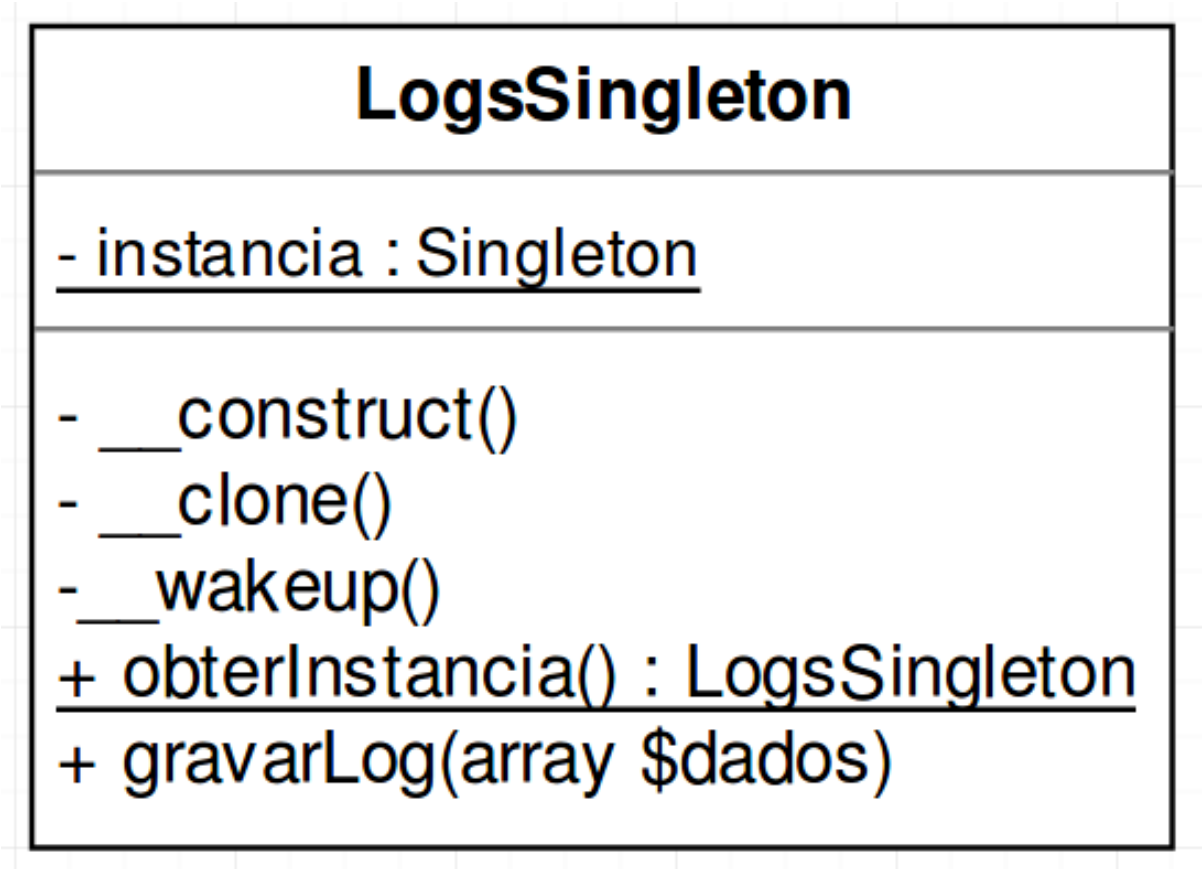

Figura 2.2: Diagrama de classes do padrão Singleton em PHP

Elementos que compõem o Singleton

O `Singleton` é uma classe responsável por controlar a criação de instâncias de si própria e por permitir que essas instâncias sejam acessadas de forma global por toda a aplicação. Esta é diferente do **Adapter**, que será visto posteriormente, já que ele é composto por um único elemento.

2.3 Resolvendo o problema dos logs com instância única

No exemplo a seguir, vamos escrever o código responsável por resolver o problema das múltiplas instâncias do sistema bancário. A ideia é criar uma classe para a gravação de logs em um arquivo de texto. A cada ocorrência de um log, esse registro será gravado no arquivo.

Essa classe seguirá o padrão **Singleton** e permitirá que apenas uma instância seja criada, não importando quantas vezes ela seja usada. No código adiante, começaremos a construir a classe `LogsSingleton`, que será a base para nosso padrão. Sugiro que crie uma pasta com o nome `singleton` e insira todos os arquivos dentro dela para efetuar o teste.

```php
<?php

namespace Singleton;

class LogsSingleton
{
    /** @var self $instancia Instância da classe de logs. */
    protected static $instancia;

    // ...

}
```

O primeiro ponto é a criação da propriedade (atributo) `$instancia`, que armazenará a instância única da classe. Como pode ser visto no exemplo, a propriedade foi definida como `static` (estática), porque, ao armazenarmos a instância em uma propriedade estática, garantimos o ponto único onde ficará a instância da classe.

A propriedade foi definida com o nível de privacidade `protected`. Desta forma, não podemos acessá-la diretamente de fora de si própria.

Conforme comentado, vamos armazenar os logs em um arquivo de texto, cujo nome será `logs.txt` e os dados ficarão em formato `JSON`. Este formato é uma estrutura simples para armazenamento e troca de dados em sistemas, que permite organizar informações de uma forma estruturada e até definir tipos de dados.

Também precisamos do método que será chamado para registrar os logs no arquivo de texto. Vamos ver como ele ficará, lembrando que o método `gravarLog` deve ficar dentro da classe `LogsSingleton`, criada anteriormente.

```
public function gravarLog(array $dados)
{
    $nomeArquivo = 'logs.txt';

    $logsAnteriores = [];
    if (filesize($nomeArquivo) > 0) {
        $conteudoDoArquivo = file_get_contents($nomeArquivo);

        $logsAnteriores = json_decode($conteudoDoArquivo, true);
    }

    $logsAnteriores[] = $dados;

    $arquivo = fopen($nomeArquivo, 'w');

    fwrite($arquivo, json_encode($logsAnteriores));
```

```
    fclose($arquivo);
}
```

O método `gravarLog` recebe um parâmetro (`$dados`) do tipo `array` . É nesse parâmetro que receberemos os dados a serem gravados no arquivo de texto.

No código a seguir, estamos declarando a variável `$logsAnteriores` como um array vazio (isso evitará erros caso o arquivo de logs esteja vazio). Depois, por meio do método `filesize` , verificamos se o arquivo de texto possui algo gravado dentro dele; se o tamanho for maior que zero, então existem dados dentro (logs).

Buscaremos o conteúdo pela função `file_get_contents` nativa do PHP, depois pegamos esse conteúdo (armazenado no formato `JSON`) para transformá-lo em um `array` com a função `json_decode` . No final, a variável `$logsAnteriores` conterá um `array` vazio caso não tenha conteúdo no arquivo, ou um `array` com os logs gravados anteriormente em outras execuções.

```
...
$logsAnteriores = [];
if (filesize($nomeArquivo) > 0) {
    $conteudoDoArquivo = file_get_contents($nomeArquivo);

    $logsAnteriores = json_decode($conteudoDoArquivo, true);
}
...
```

No código `$logsAnteriores[] = $dados;` , estamos incrementando o `array` de logs e adicionando o novo log, mesmo caso esteja vazio ou já contenha logs. Efetuamos a abertura do arquivo de texto para que possamos novamente fazer a escrita do texto na linha `$arquivo = fopen($nomeArquivo, 'w');` .

Depois, efetuamos a escrita do JSON no arquivo de texto, no código `fwrite($arquivo, json_encode($logsAnteriores));` . A função `json_encode` transforma o `array` de logs em um texto nesse formato. E para

finalizar, chamamos a função `fclose`, responsável por fechar o arquivo que havia sido aberto para escrita.

Até agora, construímos uma classe que consegue gravar logs em arquivos de texto, e possui uma variável estática e global para armazenar sua própria instância. Porém, precisamos fazer o controle dessa instância, então, vamos criar um método para isso:

```
public static function obterInstancia(): self
{
    if (empty(self::$instancia)) {
        self::$instancia = new self();
    }

    return self::$instancia;
}
```

Assim como a propriedade `$instancia`, o método `obterInstancia` também foi declarado de forma estática. Vamos chamá-lo para requisitar uma nova instância (sem instanciar a classe de logs), e somente esse método terá acesso externo à propriedade `$instancia`.

Na primeira linha dentro dele, no `if`, validamos se `$instância` está vazio com a comparação `empty(self::$instancia)` Caso esteja, isso quer dizer que nenhuma instância da classe de logs (`LogsSingleton`) foi criada ainda; se essa condição for verdadeira, então instanciamos a classe e para armazená-la com o código `self::$instancia = new self();`. O `new self()` seria o mesmo que `new LogsSingleton()`, porém, como está dentro da própria classe, podemos usá-lo assim.

No final, com `return self::$instancia;`, sempre retornamos a instância da própria classe. Na primeira vez em que ele é chamado, ele sempre vai passar dentro do `if` e instanciar a classe, porém, nas demais vezes, apenas retornará a instância criada anteriormente.

Agora, pergunto a você: o que aconteceria se executássemos o código a seguir?

```
$instanciaLogs = new LogsSingleton();
```

Ao contrário do esperado, o código funcionaria e teríamos uma nova instância de `LogsSingleton`. Isso não está de acordo com a proposta de instâncias únicas/controladas. Para evitarmos esse comportamento inesperado, devemos declarar o método construtor da classe com uma visibilidade do tipo `private` (privado):

```
private function __construct() {}
```

Desta forma, com `new LogsSingleton()`, apenas poderíamos obter a instância da classe dentro de si própria ao chamar o método `obterInstancia`. Além do construtor privado, no PHP, existem outros métodos específicos que permitem obter uma nova instância da classe, e todos devem ser declarados como privados, como mostrado a seguir:

```
private function __clone() {}
```

O método mágico `__clone` serve para fazer a clonagem de objetos.

```
private function __wakeup() {}
```

Já o método mágico `__wakeup()` serve para a desserialização de objetos.

Assim, evitamos que sejam criadas novas instâncias da classe `LogsSingleton`. Com o código desenvolvido até agora, já temos a implementação completa do padrão **Singleton**.

MÉTODOS MÁGICOS

Os métodos mágicos são funções disponíveis a partir do PHP 5. Eles foram criados quando o PHP começou a evoluir na Orientação a Objetos. Podem ser identificados por iniciar com `__` , e cada um possui uma funcionalidade específica, como o `__construct` , que serve como construtor de classes, ou o `__toString` (semelhante ao Java), chamado quando tentamos imprimir um objeto (convertê-lo em uma `string`).

Para testar essa classe, vamos criar um arquivo `index.php` e inserir o código seguinte:

```
<?php

$instancia = LogsSingleton::obterInstancia();

$novaInstancia = LogsSingleton::obterInstancia();

if ($instancia === $novaInstancia) {
    echo 'As instâncias são exatamente as mesmas!';
}
```

Tanto na primeira quanto na segunda vez que executamos `LogsSingleton::obterInstancia()` , o método vai retornar a mesma instância. Não só o valor será igual, mas ambas as variáveis (`$instancia` e `$novaInstancia`) apontarão para o mesmo endereço de memória.

A comparação `$instancia === $novaInstancia` será verdadeira e devemos ver a mensagem `"As instâncias são exatamente as mesmas!"` .

2.4 Conclusão

Além do exemplo de classe de logs apresentado, outro caso que seria muito comum é o de instâncias de uma classe de conexão com base de dados. Estas classes geralmente são utilizadas por todo o sistema.

Também vale ressaltar que existem implementações em que a classe **Singleton** permite mais de uma instância. Entretanto, sempre é um número limitado e controlado, o que ainda se enquadra com o padrão.

Apesar de todos os seus pontos negativos citados, que levam os desenvolvedores a considerarem o Singleton como um *anti-pattern*, ainda é válido ter conhecimento de seu funcionamento e sua proposta.

CAPÍTULO 3
Construindo objetos com Factory Method

TIPO: padrão de criação

Nível de dificuldade:

3.1 O problema da startup de automóveis

Imagine a seguinte situação: você trabalha como desenvolvedor em uma *startup* que tem como principal objetivo revolucionar a venda de veículos através do mundo. Os veículos vendidos na plataforma são disponibilizados por meio de acordo direto com as fabricantes. Até agora, já foram fechados contratos de venda de alguns modelos específicos, como a **Tesla** e a **Dodge**.

A ideia é desenvolver um sistema do zero que permitirá a venda online e a divulgação dos veículos através de outras plataformas (outros sistemas) de integração, entre outras funcionalidades. Repare que, só aí, já possuímos claramente dois lugares em nosso sistema que vão trabalhar diretamente com os veículos. Isso quer dizer que, se cada carro possuir uma classe específica, será necessário instanciar cada uma delas repetidas vezes, então teríamos uma repetição do código.

Além dessa repetição, que dificulta a manutenção caso precisemos alterar alguma classe de veículos, temos um problema ainda maior: não sabemos quantos carros teremos no total, pois, a cada ano, modelos saem de linha e outros são lançados (substituídos).

Agora, imagine ter de entrar em cada ponto do sistema que utiliza determinada classe de um modelo específico e alterá-la. Além da dificuldade de lembrar de todos os pontos que usam a classe, teríamos um acoplamento muito alto entre o nosso sistema e as instâncias (que podem variar ou aumentar com o tempo).

Uma solução utilizada por muitos desenvolvedores para esse tipo de problema é a aplicação do Padrão de Projeto **Factory Method**. Com ele, alteramos a parte do nosso código que possui a responsabilidade de criação de determinados objetos. Vamos encapsular essa criação, de modo que teremos um sistema mais flexível e menos acoplado, conforme será visto neste capítulo.

3.2 O padrão Factory Method

Definição

O objetivo desse padrão **Factory Method** é encapsular a criação e os objetos de um sistema. Isso ocorre por meio de outros objetos chamados de fábricas (referência à fábrica de objetos), cuja responsabilidade é apenas criar as instâncias.

No padrão **Factory Method**, temos uma interface que define um contrato para os objetos a serem criados pelas fábricas, que seriam nossos produtos finais. Também temos uma interface que define como nossas classes-fábrica deverão ser construídas.

Contratos através de interfaces

O grande benefício do uso das interfaces é estabelecer contratos entre classes. Na interface, temos apenas a definição de métodos abstratos, sem qualquer definição de código dentro. Se uma classe implementa uma interface, ela deve obrigatoriamente escrever os métodos abstratos. Porém, cada classe define o seu próprio comportamento para esse método, contanto que obedeça à assinatura definida na interface. A assinatura do método é como o identificamos de forma única, composta por elementos como o nome do método, os parâmetros e os tipos de dados que ela recebe, seu retorno etc.

Outro grande benefício em PHP é que podemos utilizar a interface em definições de tipos de dados de parâmetros de métodos ou seus retornos. Por exemplo, se criássemos uma interface `InterfaceDeTeste`, e ela fosse implementada por duas outras classes, poderíamos definir nos parâmetros da assinatura de qualquer outro método de nosso sistema essa interface. Automaticamente, esse método aceitaria ambas as classes que o implementam. Dessa forma, ao programarmos para uma interface (para seus métodos), e não para uma classe concreta, nosso sistema fica muito mais flexível.

A criação de cada um dos objetos fica encapsulada dentro de uma classe específica, responsável por isso. Todos os pontos do sistema que necessitem de instâncias dessa classe fazem solicitações para sua fábrica, dispensando o uso do operador `new`. Caso nosso produto precise ser alterado no futuro, será necessário apenas alterações em sua classe-fábrica.

Conforme a definição no livro *Design Patterns* (GAMMA; HELM; JOHNSON; VLISSIDES, 1994), os autores sugerem que o objetivo do padrão é definir uma interface para a criação dos objetos e transferir a responsabilidade de criação para as subclasses.

Com o uso deste padrão, temos uma redução do acoplamento do código. As nossas classes clientes não têm conhecimento direto das classes concretas, apenas de uma interface. Logo, temos uma maior flexibilidade, pois não estamos diretamente dependentes de um objeto concreto.

Diagrama de classes

A seguir, o diagrama de classes do padrão **Factory Method** baseado no nosso exemplo:

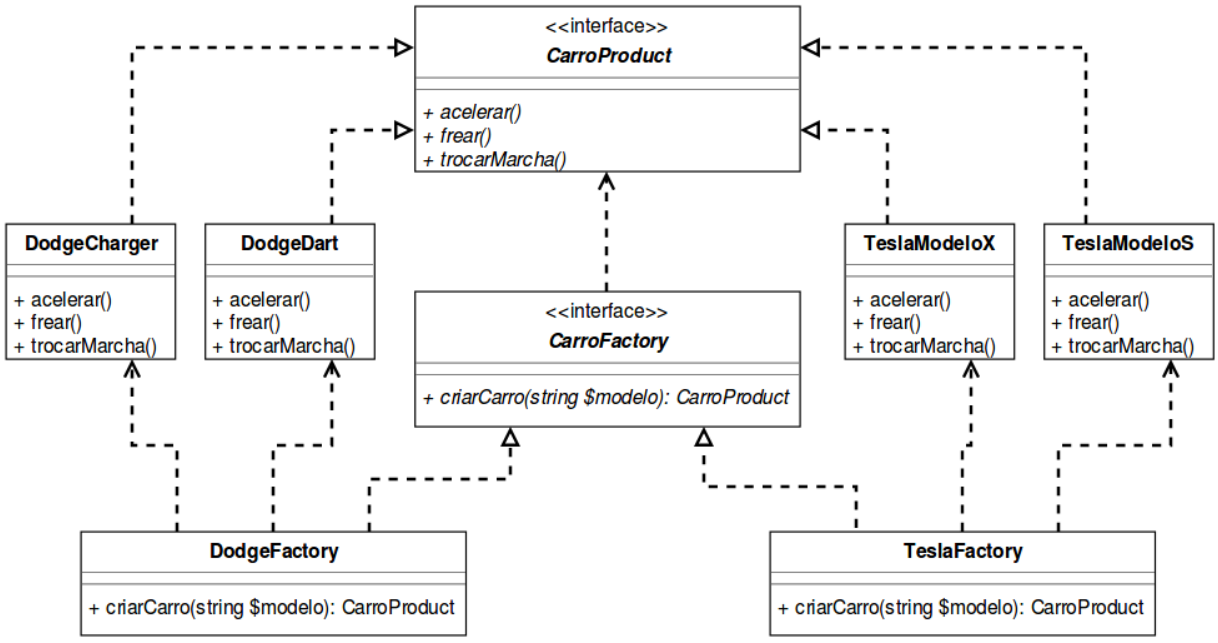

Figura 3.2: Diagrama de classes do Padrão de Projeto Factory Method

Elementos que compõem o Factory Method

- **Product** (produto abstrato): define uma interface de objetos que será criada pelo nosso método-fábrica que estará no **Concrete Creator**.
- **Concrete Product** (produto concreto): é o produto concreto que implementa a interface **Product**. Trata-se do produto final que será criado pelas fábricas.

- **Creator** (criador abstrato): é a interface que define o contrato de todas as nossas fábricas.
- **Concrete Creator** (criador concreto): é a classe concreta que implementa nossa interface **Creator**, ela representa a fábrica de objetos de cada produto.

3.3 Resolvendo o problema dos automóveis

Vamos resolver o problema da startup de automóveis com a aplicação do **Factory Method**, a primeira questão a analisarmos é qual classe de nosso sistema queremos encapsular através de nossos construtores. No nosso caso, as classes são as dos modelos de veículos.

Cada modelo diferente possuirá uma classe no sistema, sendo que os que vamos implementar inicialmente são o **Tesla Modelo S**, o **Tesla Modelo X**, o **Dodge Charger** e o **Dodge Dart**. Cada um deles corresponde aos elementos **Concrete Product** de nossa aplicação, porém, antes de criá-los, faremos a nossa interface comum a todos os produtos.

Crie uma pasta com o nome de `FactoryMethod` e, dentro dela, uma subpasta com o nome `Product`, onde ficarão todos os nossos produtos. Dentro da pasta `Product`, crie um arquivo chamado de `CarroProduct.php` e siga o exemplo:

```
<?php

namespace FactoryMethod\Product;

interface CarroProduct
{
    public function acelerar();

    public function frear();
```

```
    public function trocarMarcha();
}
```

A interface `CarroProduct` serve de contrato a todos os nossos produtos concretos (modelos de veículos). Ela possui três métodos abstratos: `acelerar`, `frear` e `trocarMarcha`. Estes serão implementados pelas classes de modelos de veículos.

Agora que já criamos nossa interface, vamos criar as classes de produtos concretos, e depois ver seu funcionamento. Todos devem ficar dentro da pasta `Product`. Começando pelos modelos da **Tesla**, crie um arquivo dentro da pasta com o nome de `TeslaModeloS.php` e, dentro do arquivo, insira o código a seguir:

```
<?php

namespace FactoryMethod\Product;

class TeslaModeloS implements CarroProduct
{
    public function acelerar()
    {
        echo "Acelerando Tesla Modelo S\n";
    }

    public function frear()
    {
        echo "Freando Tesla Modelo S\n";
    }

    public function trocarMarcha()
    {
        echo "Trocando marcha do Tesla Modelo S\n";
    }
}
```

Agora, crie um arquivo com o nome de `TeslaModeloX.php`. Vamos à sua implementação:

```php
<?php

namespace FactoryMethod\Product;

class TeslaModeloX implements CarroProduct
{
    public function acelerar()
    {
        echo "Acelerando Tesla Modelo X\n";
    }

    public function frear()
    {
        echo "Freando Tesla Modelo X\n";
    }

    public function trocarMarcha()
    {
        echo "Trocando marcha do Tesla Modelo X\n";
    }
}
```

Com os dois modelos da **Tesla**, vamos passar para os exemplos da fabricante **Dodge**. Ainda dentro da pasta `Product`, crie um arquivo com o nome de `DodgeCharger.php`, e siga o exemplo de implementação:

```php
<?php

namespace FactoryMethod\Product;

class DodgeCharger implements CarroProduct
{
    public function acelerar()
    {
        echo "Acelerando Dodge Charger\n";
    }

    public function frear()
    {
        echo "Freando Dodge Charger\n";
```

```
    }

    public function trocarMarcha()
    {
        echo "Trocando marcha do Dodge Charger\n";
    }
}
```

Para finalizar os nossos exemplos de modelos de veículos, crie um arquivo com o nome de `DodgeDart.php` e, dentro dele, insira o código a seguir:

```
<?php

namespace FactoryMethod\Product;

class DodgeDart implements CarroProduct
{
    public function acelerar()
    {
        echo "Acelerando Dodge Dart\n";
    }

    public function frear()
    {
        echo "Freando Dodge Dart\n";
    }

    public function trocarMarcha()
    {
        echo "Trocando marcha do Dodge Dart\n";
    }
}
```

Estes serão os elementos equivalentes a **Concrete Product** de nossos exemplos. No início, serão somente estas quatro classes, mas, no futuro, algum modelo pode sair de linha e novos podem entrar. Repare que todos implementam a interface `CarroProduct` : esta é a interface comum a todos eles.

Com isso, não teríamos problemas em nossa aplicação caso precisássemos futuramente substituir um modelo por outro, pois teríamos sempre os mesmos métodos: `acelerar`, `frear` e `trocarMarcha`. Note também que cada um desses métodos foi implementado à sua maneira e cada um exibe uma mensagem diferente, respectiva a seu próprio modelo.

Agora, criaremos as classes responsáveis por instanciar nossos produtos. Vamos separar as nossas classes-fábrica por fabricante de veículo. Porém, para que possamos criar as fábricas e seguir o padrão de construção de produtos, sempre criando objetos do tipo `CarroProduct`, vamos primeiro criar a interface da nossa fábrica, equivalente ao elemento **Creator** de nosso padrão.

Volte um nível de pastas e, dentro da `FactoryMethod`, crie um arquivo com o nome de `CarroFactory.php`, com o exemplo a seguir:

```
<?php

namespace FactoryMethod;

use FactoryMethod\Product\CarroProduct;

interface CarroFactory
{
    public function criarCarro(string $modeloCarro): CarroProduct;
}
```

A interface `CarroFactory` faz uso de `CarroProduct` pelo bloco de código `use FactoryMethod\Product\CarroProduct;`. O objetivo dessa interface é servir de contrato para todas as nossas classes-fábrica, os construtores concretos.

Nela, foi definido um único método, o `criarCarro`, que é o método-fábrica do padrão. Ele recebe uma `string` por parâmetro que representa o modelo do carro a ser criado – posteriormente, veremos sua utilidade. Também definimos que o retorno desse método deve obrigatoriamente retornar um objeto do tipo

`CarroProduct`, assim, garantimos a compatibilidade de nossas fábricas.

Vamos à construção de nossa primeira classe-fábrica, o nosso **Concrete Creator**. No primeiro exemplo, criaremos a fábrica de veículos da **Tesla**. Crie um arquivo com o nome de `TeslaFactory.php`, conforme a seguir:

```
<?php

namespace FactoryMethod;

use FactoryMethod\Product\{
    CarroProduct, TeslaModeloS, TeslaModeloX
};

class TeslaFactory implements CarroFactory
{
    public function criarCarro(string $modeloCarro): CarroProduct
    {
        if ($modeloCarro == 'modelo_x') {
            return new TeslaModeloX();
        } elseif ($modeloCarro == 'modelo_s') {
            return new TeslaModeloS();
        } else {
            throw new \Exception("Modelo de carro \"{$modeloCarro}\" não
existe no sistema.");
        }
    }
}
```

A classe `TeslaFactory` implementa a interface `CarroFactory`; com isso, temos obrigatoriamente a implementação do método `criarCarro`. Porém, aqui inserimos a lógica de criação de nossos veículos da **Tesla**.

Lembra-se do parâmetro `$modeloCarro`? Ele é utilizado aqui. Comparamos se ele é igual a `string` com o valor de `modelo_x`; nesse caso, retornamos a instância do `TeslaModeloX` conforme a parte do código:

```
  return new TeslaModeloX();
```

Caso o modelo seja igual a `modelo_s`, então retornamos uma instância da classe `TeslaModeloS`, conforme o código:

```
  return new TeslaModeloS();
```

Se for passado outro modelo que não existe, a verificação cairá no `else` e lançará uma exceção com uma mensagem de erro inserida dentro das classes de fábrica. Com isso, concluímos o exemplo de nossa primeira classe **Concrete Creator**.

Agora, vamos criar a responsável pelos nossos carros da **Dodge**. Crie um arquivo com o nome `DodgeFactory.php`, semelhante ao exemplo anterior mas com instâncias diferentes. Siga o exemplo de código a seguir:

```
<?php

namespace FactoryMethod;

use FactoryMethod\Product\{
    CarroProduct, DodgeCharger, DodgeDart
};

class DodgeFactory implements CarroFactory
{
    /**
     * @param string $modeloCarro
     * @return CarroProduct
     * @throws \Exception
     */
    public function criarCarro(string $modeloCarro): CarroProduct
    {
        if ($modeloCarro == 'charger') {
            return new DodgeCharger();
        } elseif ($modeloCarro == 'dart') {
            return new DodgeDart();
        } else {
            throw new \Exception("Modelo de carro \"{$modeloCarro}\" não
existe no sistema.");
```

```
        }
    }
}
```

A classe `DodgeFactory` também implementa a interface `CarroFactory`, porém seu método `criarCarro` é um pouco diferente. As opções de `string` que ele aceita em sua criação são: `charger`, que retorna uma instância da classe `DodgeCharger`; e a string `dart`, que retorna uma instância de `DodgeDart`. Também temos uma opção caso algum modelo inexistente seja passado por parâmetro.

Um ponto importante a ser observado é que a passagem de uma `string` com as opções de objetos a serem criadas deve ser usada com cuidado, pois não existe uma grande segurança de que os clientes que solicitarem a criação de objetos passarão os valores corretos. Além disso, pode haver uma dificuldade de documentar esses valores ou de um entendimento futuro do que cada um significa.

Em outras linguagens orientadas a objetos, como o **Java**, é comum o uso de objetos **Enumeration**. Com eles, conseguimos definir os tipos disponíveis de uma forma mais elegante. Deixo aqui como sugestão que pesquise o funcionamento da biblioteca *eloquent/enumeration* (https://github.com/eloquent/enumeration), que possibilita a criação de classes que simulam o comportamento de um **Enumeration**.

Ao pesquisar sobre o padrão **Factory Method** na internet ou em livros, é possível encontrar três variações nos exemplos dessa classe-fábrica. A primeira é a de que cada classe **Concrete Creator** é responsável por construir um único tipo de objeto – diferente do segundo exemplo, em que agrupamos os objetos por categoria (fabricante do veículo). O outro exemplo encontrado é o que agrupa a construção de todos os objetos dentro de uma única classe-fábrica.

Na primeira opção, temos uma divisão maior das responsabilidades de criação – nesse caso, teríamos classes `DodgeChargeFactory`,

`DodgeDartFactory`, `TeslaModeloSFactory` e `TeslaModeloXFactory`. O único problema que podemos observar nessa solução é que teríamos muitas classes, assim moveríamos o operador condicional que colocamos dentro do método `criarCarro` para fora das classes de criação.

Nossas classes clientes assumiriam essa responsabilidade total de decidir qual fábrica elas necessitam, e esse código poderia se repetir em muitos pontos de nosso sistema. Cada vez que surgisse um novo modelo, teríamos de alterar tudo. Essa forma é muito utilizada para a aplicação do padrão, assim como a que optamos.

Uma outra alternativa (menos usada) seria implementar uma única classe-fábrica para todos os veículos. Ela teria uma condição com um `if` semelhante ao de dentro do método `criarCarro`, ou até um `switch` com todas as opções de modelos de carro. Porém, imagine quantas linhas de código poderíamos ter dentro dessa classe. A cada novo veículo, teríamos mais uma condição!

A alternativa pela qual optei para este exemplo foi o meio-termo entre as duas anteriores. Os operadores condicionais foram quebrados através de categorias de veículos, e cada fabricante possui sua classe-fábrica com os modelos agrupados. Assim, não transferimos toda a responsabilidade de qual classe instanciar para o cliente, e também temos um código mais separado e estruturado com uma divisão lógica.

Para testarmos nossas classes, crie um arquivo com o nome de `index.php` e siga o exemplo:

```php
<?php

require_once('../autoloader.php');

$teslaFactory = new \FactoryMethod\TeslaFactory();

$dodgeFactory = new \FactoryMethod\DodgeFactory();
```

Primeiro, estamos criando uma instância de `TeslaFactory` e atribuindo a variável `$teslaFactory`. Depois, criamos uma instância de `DodgeFactory` e atribuímos a variável `$dodgeFactory`. Até aqui, temos as nossas duas fábricas criadas.

Agora, vamos utilizar essa classe para construir nossos objetos. Insira o código a seguir no arquivo:

```
try {
    $teslaModeloS = $teslaFactory->criarCarro('modelo_s');
    $teslaModeloX = $teslaFactory->criarCarro('modelo_x');

    $dodgeCharger = $dodgeFactory->criarCarro('charger');
    $dodgeDart = $dodgeFactory->criarCarro('dart');
} catch (Exception $e) {
    echo $e->getMessage();
}
```

O uso das classes-fábricas para a criação das instâncias de veículos foi inserido dentro de uma cláusula `try`.

Lembre-se de que essas classes podem lançar exceções de erro caso seja informado um nome de modelo de veículos inexistente. Caso isso ocorra, o erro será capturado pela cláusula `catch`, assim exibiremos a mensagem de erro pela linha de código `echo $e->getMessage();`. Trataremos o nosso erro de aplicação de uma forma mais elegante do que um *Fatal error*, que ocorre no PHP quando não existe tratamento.

Dentro da clausula `try`, fazemos a chamada aos métodos-fábrica `criarCarro`, como podemos ver na primeira e segunda linha:

```
    $teslaModeloS = $teslaFactory->criarCarro('modelo_s');
    $teslaModeloX = $teslaFactory->criarCarro('modelo_x');
```

Usamos a classe `TeslaFactory` para construir os objetos de produtos. Na primeira linha, retornamos um objeto do tipo `TeslaModeloS` e, na segunda, é retornando um objeto do tipo `TelaModeloX`.

Depois, utilizamos a classe `DodgeFactory` para construir os modelos da **Dodge**:

```
$dodgeCharger = $dodgeFactory->criarCarro('charger');
$dodgeDart = $dodgeFactory->criarCarro('dart');
```

Primeiro, criamos um objeto do tipo `DodgeCharger` e, na linha seguinte, a nossa fábrica constrói um objeto do tipo `DodgeDart`. A única diferença para construir um novo tipo de objeto é o parâmetro passado para as classes-fábrica.

Para testarmos se nossos objetos de veículos foram construídos de forma correta, vamos chamar os métodos `acelerar`, `frear` e `trocarMarcha` de nossos produtos concretos, especificados na interface `CarroProduct`. Insira no arquivo as linhas a seguir:

```
echo $teslaModeloS->acelerar();
echo $teslaModeloS->frear();
echo $teslaModeloS->trocarMarcha();
echo "\n";

echo $dodgeCharger->acelerar();
echo $dodgeCharger->frear();
echo $dodgeCharger->trocarMarcha();
echo "\n";

echo $teslaModeloX->acelerar();
echo $teslaModeloX->frear();
echo $teslaModeloX->trocarMarcha();
echo "\n";

echo $dodgeDart->acelerar();
echo $dodgeDart->frear();
echo $dodgeDart->trocarMarcha();
```

Se executarmos o arquivo `index.php`, a saída esperada deve ser a seguinte:

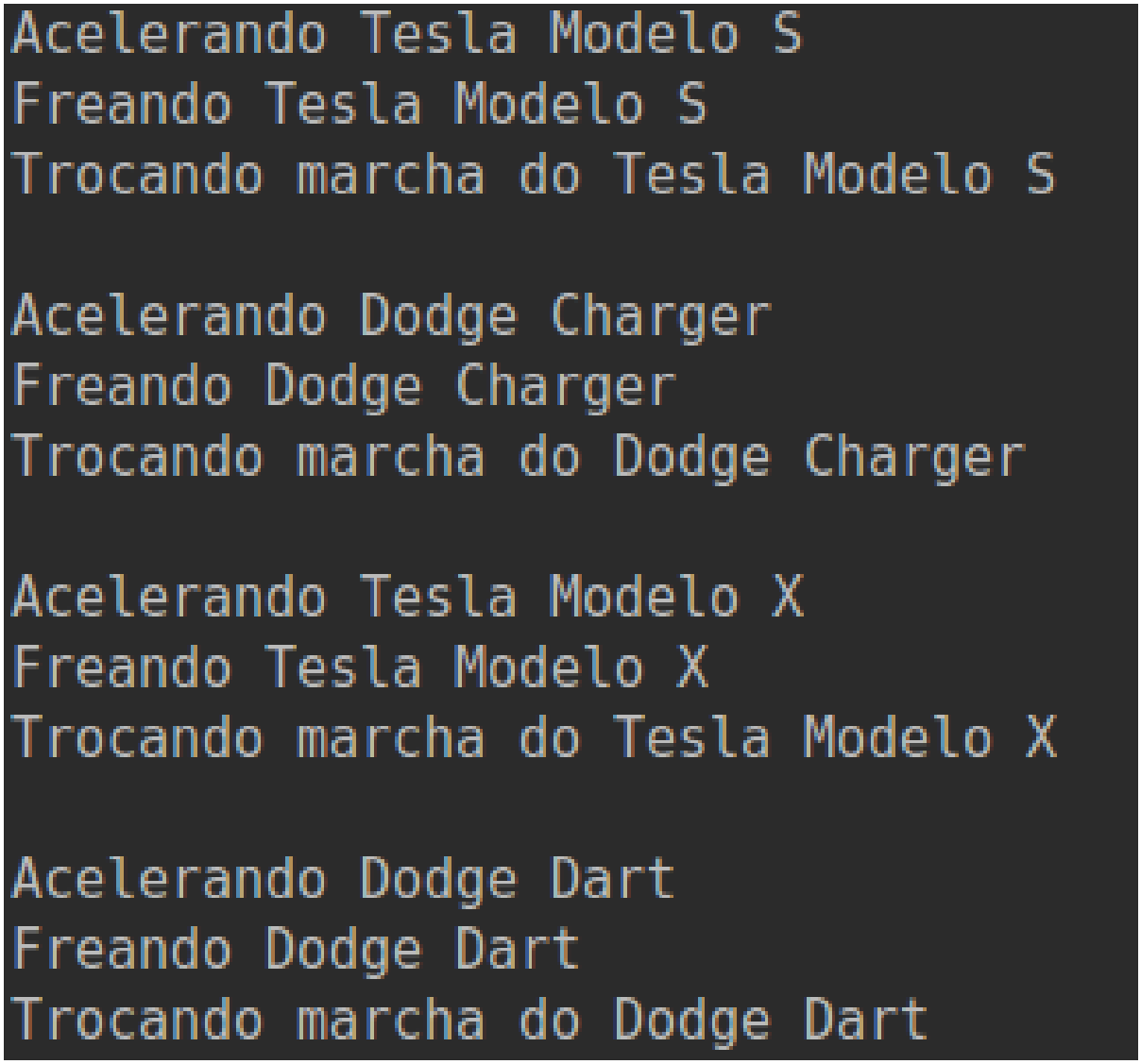
Acelerando Tesla Modelo S
Freando Tesla Modelo S
Trocando marcha do Tesla Modelo S
Acelerando Dodge Charger
Freando Dodge Charger
Trocando marcha do Dodge Charger
Acelerando Tesla Modelo X
Freando Tesla Modelo X
Trocando marcha do Tesla Modelo X
Acelerando Dodge Dart
Freando Dodge Dart
Trocando marcha do Dodge Dart

Figura 3.3: Resultado esperado para o teste do Factory Method

A estrutura de arquivos ficou desta forma:

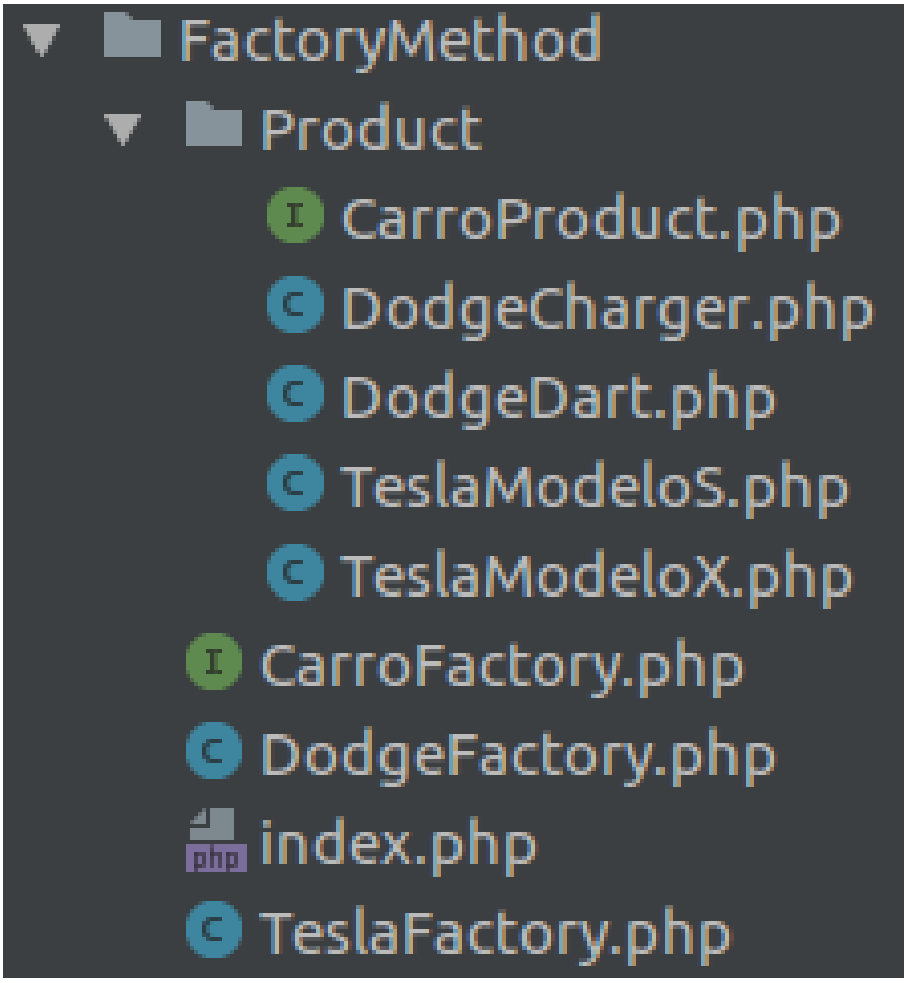

Figura 3.4: Estrutura de arquivos da implementação do Padrão de Projeto Factory Method

3.4 Conclusão

O Padrão de Projeto **Factory Method** define uma interface padronizadora para produtos (objetos) do sistema que queremos criar, transferindo essa responsabilidade de criação para as subclasses. Desta forma, temos um nível maior de abstração das classes de produtos concretos, e é possível criar esses objetos sem especificar qual classe será usada.

CAPÍTULO 4
Prototipação de objetos com Prototype

TIPO: padrão de criação

Nível de dificuldade:

4.1 O problema da editora de livros

Recentemente, a editora de livros na qual você trabalha como desenvolvedor teve uma ideia inovadora. Ela passará a receber solicitações de livros com dedicatória para um nome de pessoa à escolha do comprador. Este nome sairá impresso na capa do livro, abaixo do título, e os livros serão impressos todos juntos pela aplicação.

A ideia é ter uma listagem de nomes para determinado livro, e o esperado por lote é ter milhares de nomes. Ao pensarem em uma solução, a equipe de desenvolvedores sabia que existiriam objetos no sistema para representar cada livro, contendo atributos com todas as suas informações, entre elas: título, categoria do livro etc. Bastaria ter uma classe para estes livros e instanciá-la diversas vezes em um laço de repetições. Porém, em cada uma das milhares de voltas desse laço, teríamos de definir todas as informações do livro – que são exatamente iguais (título, assunto etc.) – e apenas alterar o nome da dedicatória. Após isso, seria só mandar para a impressão.

A equipe desenvolveu essa solução, e começou a receber os pedidos e armazenar os nomes em seu sistema. Só no primeiro mês, ela recebeu um número enorme de solicitações, muito acima do esperado, e foi aí que perceberam o grande problema: devido à grande quantidade de livros que deveriam ser impressos, sempre que a impressão começava, o uso de memória do servidor ia até seu limite e o programa parava.

Depois de algumas análises, descobriram que o programa de impressão estava usando muita memória do servidor, pois, no nível da aplicação, havia muitas classes que eram instanciadas e, nelas, muitas propriedades iguais eram inseridas em seus atributos através de funções *setter*. Após mais algumas análises de solução do novo problema, chegaram à conclusão de que aplicação do Padrão de Projeto **Prototype** era perfeita para resolver este problema de memória do servidor.

4.2 O padrão Prototype

Definição

O objetivo desse padrão é definir alguns objetos que servem como uma instância de protótipo, clonada para a criação de novos objetos. Diferente dos demais padrões de criação, o **Prototype** utiliza o próprio objeto para criar as novas instâncias em vez de usar classes auxiliares para fazer isso.

Este padrão é indicado principalmente quando se quer criar cópias exatas de um objeto, incluindo seus atributos. Assim, temos um custo muito menor de memória do que instanciar o mesmo objeto muitas vezes e, em cada uma dessas vezes, setar exatamente os mesmos valores em suas propriedades. Aliás, também é interessante seu uso nos casos em que temos poucas variações. Ainda assim, teríamos um ganho na performance.

Diagrama de classes

A seguir, veja o diagrama de classes do padrão **Prototype**:

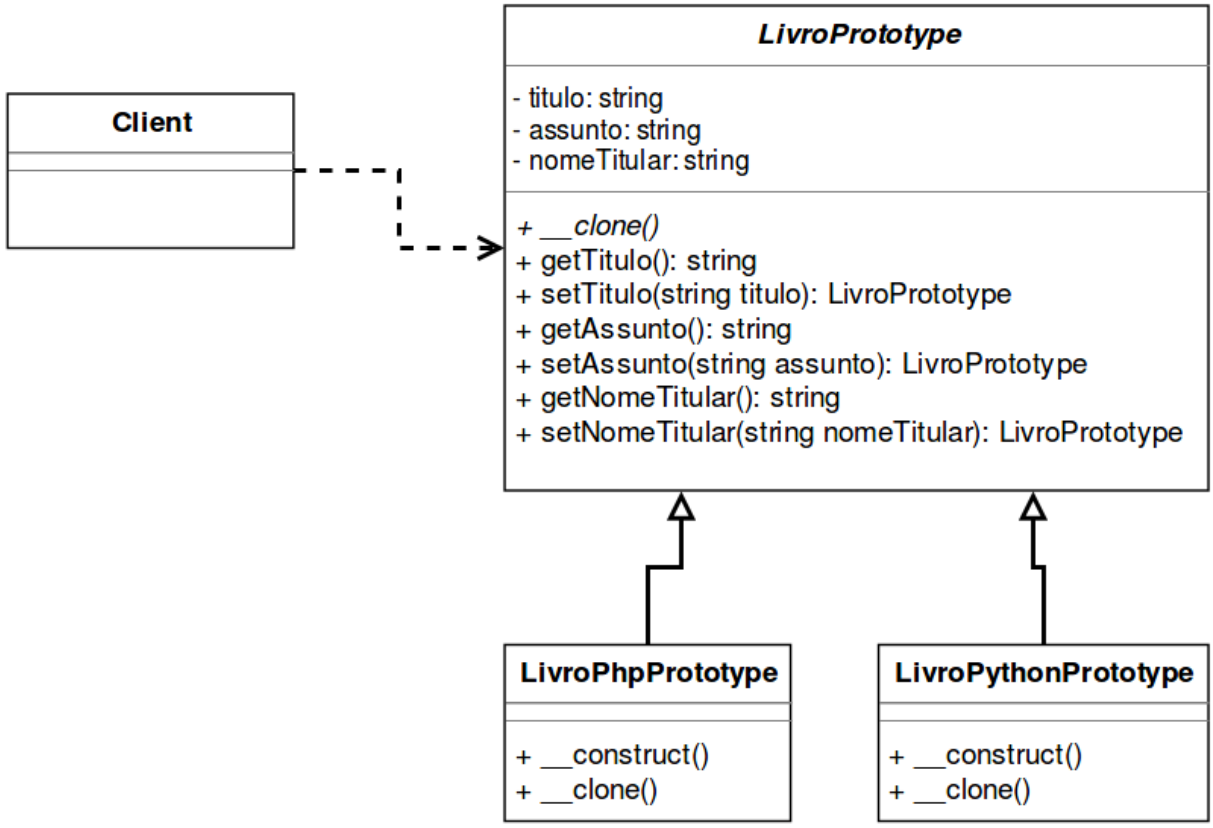

Figura 4.2: Diagrama de classes do Padrão de Projeto Prototype

Elementos que compõem o Prototype

- **Prototype** (protótipo): uma classe abstrata que define a interface com os métodos que a permitem se autoclonar.
- **Concrete Prototype** (protótipo concreto): classe concreta que implementa a interface `Prototype`.
- **Client** (cliente): classe responsável por criar os objetos de protótipo e cloná-los por meio de seus métodos.

4.3 Resolvendo o problema da impressão de livros

A implementação do Padrão de Projeto **Prototype** é muito simples quando comparado a outros padrões de criação. Para resolvermos o problema da impressão dos livros com a clonagem de objetos, primeiro vamos iniciar desenvolvendo nossa interface de protótipo que servirá de contrato. Assim, vamos garantir que todas as nossas classes de livros possam ser clonadas.

Crie uma pasta com o nome de `Prototype` e, dentro dela, crie um arquivo com o nome de `LivroPrototype.php`. Segue o exemplo da implementação desta classe:

```php
<?php

namespace Prototype;

abstract class LivroPrototype
{
    protected $titulo;

    protected $assunto;

    protected $nomeTitular;

    abstract public function __clone();

    public function getTitulo(): string
    {
        return $this->titulo;
    }

    public function setTitulo(string $titulo): LivroPrototype
    {
        $this->titulo = $titulo;
        return $this;
    }
```

```
    public function getAssunto(): string
    {
        return $this->assunto;
    }

    public function setAssunto(string $assunto): LivroPrototype
    {
        $this->assunto = $assunto;
        return $this;
    }

    public function getNomeTitular(): string
    {
        return $this->nomeTitular;
    }

    public function setNomeTitular(string $nomeTitular): LivroPrototype
    {
        $this->nomeTitular = $nomeTitular;
        return $this;
    }
}
```

A classe `LivroPrototype` foi criada como `abstract`, e ela será estendida pelas classes de protótipo concretas. Desta forma, além de servir como interface para seu método abstrato `__clone` (será explicado posteriormente), ela também possuirá métodos concretos que serão reaproveitados por todas as suas subclasses.

Essa classe possui três atributos que também serão compartilhados por todas as suas subclasses: `$titulo`, que é do tipo `string` e serve para armazenar o título dos livros; `$assunto`, que também é do tipo `string` e serve para armazenar um texto com uma espécie de categoria do livro; e `$nomeTitular`, que serve para armazenar o nome do destinatário que será inserido na capa do livro.

Para cada uma dessas propriedades, temos um método *getter* e um *setter*. Os métodos são `getTitulo`, `setTitulo`, `getAssunto`, `setAssunto`, `getNomeTitular` e `setNomeTitular`.

MÉTODOS GETTER E SETTER

Estes métodos são usados como seletores e modificadores de propriedades de uma classe. Eles têm como objetivo encapsular os dados do objeto, evitando que sejam acessados diretamente. Assim, podemos aplicar tratamentos ou até validações antes de qualquer alteração ou solicitação desses dados.

Além desses métodos, temos também a declaração do método abstrato `__clone`, que também é mágico, como explicado no capítulo anterior. Ele é invocado automaticamente quando clonamos um objeto utilizando a palavra reservada `clone`. Ele foi declarado como `abstract`, pois queremos que sua implementação seja feita nas subclasses que estendem essa classe.

Agora, vamos implementar o nosso primeiro exemplo de um protótipo concreto, uma classe de protótipo que será clonada por nosso cliente. Ela representará um objeto de livro de nosso sistema quando instanciada. Crie um arquivo com o nome de `LivroPhpPrototype.php` e siga o exemplo:

```php
<?php

namespace Prototype;

class LivroPhpPrototype extends LivroPrototype
{
    public function __construct()
    {
        $this->setAssunto('PHP');
    }

    public function __clone()
    {
        echo "Livro PHP Clonado\n";
    }
}
```

A classe `LivroPrototype` representa o nosso protótipo. Ela estende da classe abstrata `LivroPrototype`, herdando assim suas propriedades e métodos. Em seu método construtor (`__construct`), definimos o assunto do livro ao chamar o método `setAssunto`, herdado de sua superclasse. Passamos também uma `string` com o valor `PHP`, o que quer dizer que os objetos clonados dessa classe serão livros que tratam sobre a linguagem PHP. Este será um valor padrão para esses objetos.

O segundo método declarado na classe é o método mágico `__clone`, definido como `abstract` na classe `LivroPrototype`. Conforme falado anteriormente, ele é invocado automaticamente quando usamos a palavra reservada `clone` do PHP para clonar um objeto.

É importante salientar que esse método não precisa ser obrigatoriamente declarado para que um objeto seja clonado, porém vamos implementá-lo para testar o disparo de uma ação ao efetuar a clonagem. Dentro desse método, inserimos um `echo` para exibir a mensagem `Livro PHP Clonado`.

Com isso, basicamente já temos a base de nosso padrão. Então, vamos construir o nosso exemplo de cliente que ficará responsável por manipular e efetuar a clonagem desses objetos. Crie um arquivo com o nome de `index.php` e comece inserindo o código a seguir:

```
<?php

require_once('../autoloader.php');

$compradores = [
    'Gabriel Anhaia',
    'Anderson Scherer',
    'Braian Ottoni'
];

$livroPhp = new \Prototype\LivroPhpPrototype();
$livroPhp->setTitulo('Padrões de Projeto em PHP 7.2');
```

Nas linhas iniciais, estamos declarando a variável `$compradores` que armazenará um `array` com a lista de nome dos compradores dos livros – aqueles que serão usados para a customização. Existem apenas três nomes, mas poderiam ser milhões, conforme comentado em nosso problema no início do capítulo.

Depois, estamos declarando a variável `$livroPhp`, que recebe uma instância de nossa classe de protótipo. Após a declaração, chamamos o método `setTitulo` passando o valor `Padrões de Projeto em PHP 7.2`, que será o título dos livros clonados. Esse objeto não possui qualquer nome de titular definido na propriedade `$nomeTitular`.

A partir dessa instância, teremos um objeto pronto para ser clonado. Agora, insira as seguintes linhas de código no mesmo arquivo:

```
$livros = [];

foreach ($compradores as $nomeComprador) {
    $livroComprador = clone $livroPhp;
    $livroComprador->setNomeTitular($nomeComprador);

    $livros[] = $livroComprador;
}
```

A variável `$livros` vai armazenar todos os nossos livros já com os nomes alterados. No `foreach`, estamos percorrendo a lista de compradores para criar um livro customizado para cada nome de pessoa. A cada volta do *loop*, estamos declarando a variável `$livroComprador` e usando a palavra reservada `clone` junto ao objeto que queremos clonar (`$livroPhp`).

Com isso, temos uma instância de `LivroPhpPrototype` idêntica à armazenada dentro da variável `$livroPhp`, entretanto, é uma referência diferente na memória. Se alterarmos um objeto, o outro não muda.

Na segunda linha dentro do `foreach`, definimos o nome do titular do objeto clonado pelo método `setNomeTitular`. No final, atribuímos o

livro clonado ao final da lista de `$livros`. Para finalizar o exemplo, insira o código a seguir:

```
print_r($livros);
```

Ao executar esse código, teremos a seguinte saída:

```
Livro PHP Clonado
Livro PHP Clonado
Livro PHP Clonado
<pre>Array
(
    [0] => Prototype\LivroPhpPrototype Object
        (
            [titulo:protected] => Padrões de Projeto em PHP 7.2
            [assunto:protected] => PHP
            [nomeTitular:protected] => Gabriel Anhaia
        )

    [1] => Prototype\LivroPhpPrototype Object
        (
            [titulo:protected] => Padrões de Projeto em PHP 7.2
            [assunto:protected] => PHP
            [nomeTitular:protected] => Anderson Scherer
        )

    [2] => Prototype\LivroPhpPrototype Object
        (
            [titulo:protected] => Padrões de Projeto em PHP 7.2
            [assunto:protected] => PHP
            [nomeTitular:protected] => Braian Peres Ottoni
        )

)
```

Figura 4.3: Resultado esperado para o teste do Prototype

Como podemos notar, o objeto armazenado dentro de `$livroPhp` foi usado apenas para a clonagem. No final, temos todos os clones com suas diferentes propriedades (nome do comprador) dentro da variável `$livros`; entretanto, não precisamos modificar o título ou o assunto a cada novo livro criado.

A estrutura de arquivos ficou conforme a figura seguinte:

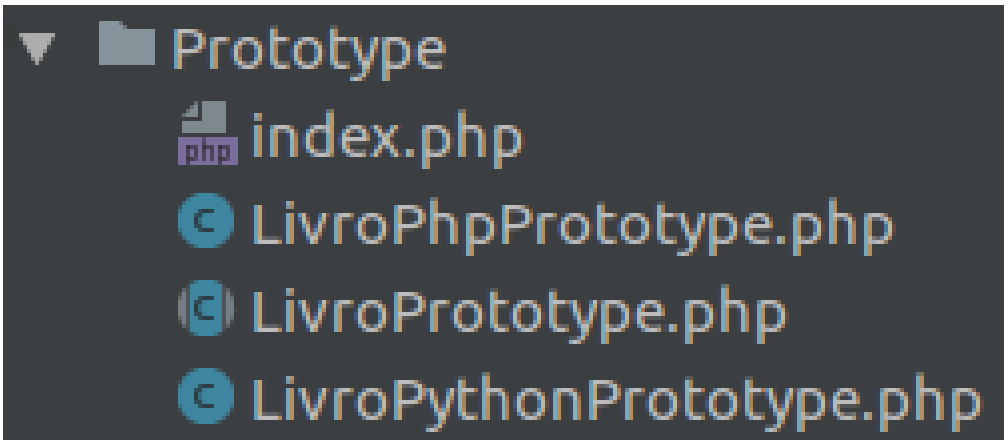

Figura 4.4: Estrutura de arquivos da implementação do Padrão de Projeto Prototype

4.4 Conclusão

Ao analisarmos o Padrão de Projeto **Prototype**, podemos concluir que ele é uma excelente alternativa, oferecendo as mesmas vantagens de outros. Ele oculta os produtos criados de seus solicitantes (clientes) e reduz consideravelmente o acoplamento de nossa aplicação. Assim, mantém um contrato através de sua interface que aumenta a flexibilidade nas alterações de nossos produtos (objetos clonados).

Seu uso é recomendado quando a criação de objetos torna-se um processo muito caro. Com ele, aproveitamos propriedades compartilhadas entre um conjunto de objetos, e evitamos o uso do operador `new`.

Capítulo 5
Objetos complexos com Builder

Tipo: padrão de criação

Nível de dificuldade:

5.1 O problema da fábrica de foguetes

Atualmente, uma grande empresa que constrói nada menos do que foguetes espaciais teve a necessidade de um software para a gestão dos seus produtos, e você foi contratado para executar este serviço. O sistema basicamente permitiria o cadastro dos foguetes que a empresa disponibiliza. Posteriormente, os foguetes previamente cadastrados passariam por diversas rotinas de teste através do mesmo sistema usado como catálogo para apresentação e venda.

Cada foguete possui suas respectivas características: seu modelo, um tipo de motor, tanque de combustível, número de assentos para os tripulantes, entre uma infinidade de outras características específicas de cada modelo.

Todos os modelos dentro do sistema possuirão uma classe em comum, porém, os atributos de cada um são diferentes e devem ser informados quando criamos cada um dos objetos que os representa. Por exemplo, um foguete **Modelo I** possui 3 motores, já o foguete **Modelo II** possui 5; ou seja, são características particulares de cada

um. Além disso, temos uma série de outros atributos que definem suas representações.

Cada objeto, representante de um foguete, possui uma montagem complexa na qual devemos atribuir diversos valores a suas propriedades. Precisamos simplificar e abstrair essa criação de tal forma que seja fácil construir esses diferentes foguetes. Você, como desenvolvedor experiente, sabe que o padrão **Builder** é perfeito para solucionar este problema.

5.2 O padrão Builder

Definição

O Padrão de Projeto **Builder** tem como objetivo simplificar a construção de objetos complexos, abstraindo essa complexidade por meio de classes específicas. Geralmente, temos complexidade na construção de um objeto quando ele possui um número muito grande de propriedades. Ele separa a construção desses objetos de suas representações, de modo que seja possível predefini-las.

Assim como o padrão *Factory*, o **Builder** encapsula a criação de objetos, ocultando os detalhes de sua criação e diminuindo o acoplamento do código. Seu foco é a divisão de responsabilidades na construção de um objeto, definindo os passos que devem ser executados. Já no *Factory*, não existe esse passo a passo que simplifica a criação dos objetos, nem a separação de suas criações com as representações.

Diagrama de classes

A seguir, veja o diagrama de classes do padrão **Builder**:

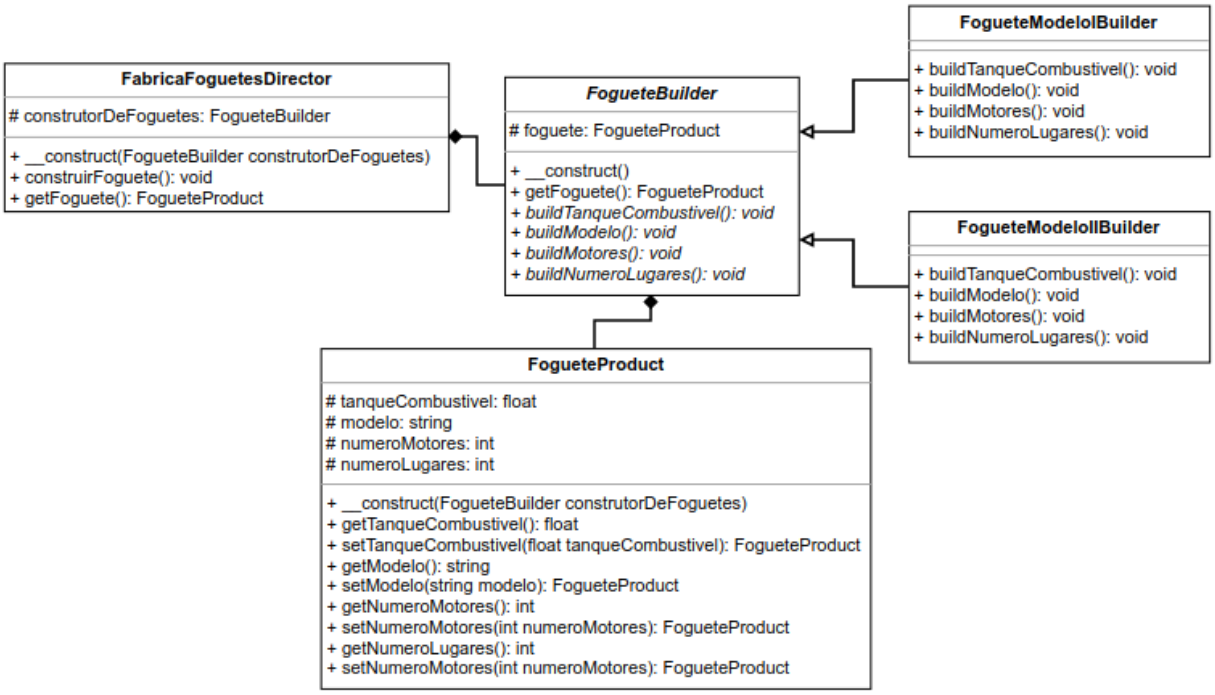

Figura 5.2: Diagrama de classes do Padrão de Projeto Builder

Elementos que compõem o Builder

- **Product** (produto): é o objeto complexo construído pelo `Builder`.
- **Builder** (construtor): é uma interface que serve como contrato para a construção das partes do nosso objeto `Product`.
- **Concrete Builder** (construtor concreto): a implementação concreta que constrói o `Product` de acordo com os dados especificados nela.
- **Director**: é a classe responsável por definir a construção dos objetos utilizando a interface `Builder`.

5.3 Resolvendo o problema dos foguetes

Para que possamos construir o nosso exemplo, vamos começar criando uma pasta com o nome de `Builder` e, dentro dela, um

arquivo com o nome de `FogueteProduct.php` , seguindo o exemplo:

```php
<?php

namespace Builder;

class FogueteProduct
{
    protected $tanqueCombustivel;

    protected $modelo;

    protected $numeroMotores;

    protected $numeroLugares;

    public function __toString(): string
    {
        $result = "Foguete model: {$this->getModelo()}\n";
        $result .= "Litros do tanque: {$this->getLitrosCombustivel()}\n";
        $result .= "Número de motores: {$this->getNumeroMotores()}\n";
        $result .= "Número de lugares: {$this->getNumeroLugares()}\n";

        return $result;
    }

    public function getLitrosCombustivel(): float
    {
        return $this->litrosCombustivel;
    }

    public function setLitrosCombustivel(float $litrosCombustivel):
FogueteProduct
    {
        $this->litrosCombustivel = $litrosCombustivel;
        return $this;
    }

    public function getModelo(): string
    {
        return $this->modelo;
```

```
    }

    public function setModelo(string $modelo): FogueteProduct
    {
        $this->modelo = $modelo;
        return $this;
    }

    public function getNumeroMotores(): int
    {
        return $this->numeroMotores;
    }

    public function setNumeroMotores(int $numeroMotores): FogueteProduct
    {
        $this->numeroMotores = $numeroMotores;
        return $this;
    }

    public function getNumeroLugares(): int
    {
        return $this->numeroLugares;
    }

    public function setNumeroLugares(int $numeroLugares): FogueteProduct
    {
        $this->numeroLugares = $numeroLugares;
        return $this;
    }
}
```

A classe `FogueteProduct` representa o elemento `Product` de nosso padrão. Ela possui 4 atributos que definem o foguete com o qual estamos trabalhando:

- `$tanqueCombustivel` – Um dado do tipo `float` que define quantos litros de combustível o tanque comporta.
- `$modelo` – Uma `string` com o modelo do foguete.
- `$numeroMotores` – Um dado do tipo `int` que define o número de motores do foguete.

- `$numeroLugares` – Também é um dado do tipo `int` que define quantos lugares para passageiros o foguete possui.

Todos os atributos possuem um método *getter* e um *setter* para o controle de acesso dos dados (encapsulamento das propriedades). Se pararmos para analisar esta classe, ela é considerada um **DTO** (*Data Transfer Object*).

> **O QUE É DTO?**
>
> O termo **DTO** refere-se a objetos criados apenas para o transporte de dados entre as camadas de nossa aplicação. Note que não existem regras de negócio na nossa classe de exemplo `FogueteProduct`. Um **DTO** apenas representa e transporta dados.

Temos um método um pouco incomum até agora em nossos exemplos: estamos definindo o método `__toString` em nossa classe. Este é mais um na listagem dos métodos mágicos do PHP, e quem já programou em **Java** deve tê-lo utilizado.

Ele é invocado automaticamente quando tentamos converter nosso objeto em uma `string`. O exemplo seguinte mostra isso:

```
<?php

$foguete = new FogueteProduct();
$foguete->setModelo('Test')
  ->setNumeroLugares(5)
  ->setNumeroMotores(2)
  ->setLitrosCombustivel(500);

echo $foguete;
```

Caso não tivéssemos definido o método `__toString`, seria apresentado o erro: `Object of class FogueteProduct could not be converted to string on line number X`, que diz que ele não pode ser convertido para uma `string`. Quando definimos o método, geramos uma saída de texto que representa nosso objeto. No nosso caso, ela

será usada para a exibição final de cada foguete de forma simples (apenas com um `echo`).

Agora, dando continuidade à implementação do padrão, antes de construir nossos elementos `Builder` concretos, precisamos definir a interface que cada um deles vai seguir. Essa mesma interface será utilizada posteriormente.

Crie um arquivo com o nome de `FogueteBuilder.php` e, dentro do arquivo, insira o código a seguir:

```php
<?php

namespace Builder;

abstract class FogueteBuilder
{
    protected $foguete;

    public function __construct()
    {
        $this->foguete = new FogueteProduct();
    }

    public function getFoguete(): FogueteProduct
    {
        return $this->foguete;
    }

    abstract public function buildTanqueCombustivel();

    abstract public function buildModelo();

    abstract public function buildMotores();

    abstract public function buildNumeroLugares();
}
```

A classe `FogueteBuilder` é do tipo `abstract`, e será estendida por cada construtor concreto (**Concrete Builder**) que implementarmos.

Cada `Builder` montará uma instância de `FogueteProduct`, que ficará armazenada na propriedade `$foguete` de nossa classe. Essa propriedade é setada dentro de nosso método construtor, pelo código `$this->foguete = new FogueteProduct();`. Ou seja, quando instanciarmos uma classe `Builder`, teremos automaticamente um foguete pronto para ser montado.

O segundo método concreto dessa classe é o `getFoguete`, que serve apenas para retornar o nosso foguete montado pelo `Builder`. Depois, temos os métodos abstratos `buildTanqueCombustivel`, `buildModelo`, `buildMotores` e `buildNumeroLugares`, que serão implementados obrigatoriamente em nossas classes concretas. Cada um deles ficará responsável por preencher uma propriedade do nosso foguete, sendo que as propriedades vão variar de acordo com o construtor que usarmos – um para cada modelo de foguete.

Vamos construir nossos exemplos de construtores concretos (**Concrete Builder**). O primeiro será o responsável pelo **Foguete Modelo I**, então crie um arquivo com o nome de `FogueteModeloIBuilder.php`, conforme o exemplo:

```php
<?php

namespace Builder;

class FogueteModeloIBuilder extends FogueteBuilder
{
    public function buildTanqueCombustivel()
    {
        $this->foguete->setLitrosCombustivel(1000);
    }

    public function buildModelo()
    {
        $this->foguete->setModelo('Foguete Modelo I');
    }

    public function buildMotores()
    {
```

```
        $this->foguete->setNumeroMotores(3);
    }

    public function buildNumeroLugares()
    {
        $this->foguete->setNumeroLugares(8);
    }
}
```

A classe `FogueteModeloIBuilder` estende de `FogueteBuilder`. Quando essa classe for instanciada, teremos uma instância de nosso produto por meio do construtor da superclasse. Lembre-se de que o objeto estará na propriedade `$foguete`.

Nessa classe, foram implementados os métodos abstratos da superclasse. Dentro de cada um deles, definimos uma parte da construção de nosso foguete. O modelo é **Foguete Modelo I**, e possui **1000** litros de capacidade em seu tanque de combustível, **3** motores e **8** lugares para passageiros. Este será o construtor utilizado para construir apenas **Foguetes Modelo I**, definindo suas características únicas.

Agora vamos criar o nosso segundo exemplo de construtor concreto (**Concrete Builder**). Ele será responsável por montar os **Foguetes Modelo II**. Crie um arquivo com o nome `FogueteModeloIIBuilder.php` conforme o exemplo:

```
<?php

namespace Builder;

class FogueteModeloIIBuilder extends FogueteBuilder
{
    public function buildTanqueCombustivel()
    {
        $this->foguete->setLitrosCombustivel(850);
    }

    public function buildModelo()
    {
```

```
        $this->foguete->setModelo('Foguete Modelo II');
    }

    public function buildMotores()
    {
        $this->foguete->setNumeroMotores(2);
    }

    public function buildNumeroLugares()
    {
        $this->foguete->setNumeroLugares(6);
    }
}
```

A classe `FogueteModeloIIBuilder` é muito parecida com `FogueteModeloIBuilder`, exceto pela implementação na hora de construir as partes de nosso foguete. O **Foguete Modelo II** possui o limite de `850` litros de combustível em seu tanque, `2` motores e `6` lugares para a tripulação.

Conforme citei na definição do padrão no início do capítulo, estamos separando as etapas da construção de nosso objeto em pequenas partes (métodos), assim, diminuímos a complexidade da sua criação. Em nosso exemplo, definimos apenas um valor, porém, dependendo da situação, poderíamos ter regras de cálculos ou alguma lógica específica para a definição dessas características de um produto. Por exemplo, poderíamos ter uma propriedade que define o preço do foguete, que seria calculada de acordo com os índices da bolsa de valores.

Vamos criar a classe responsável por fazer a chamada de cada um desses métodos de nosso `Builder`, responsáveis por montar cada parte de nosso objeto. Ela corresponde ao elemento `Director` e será chamada diretamente pelo nosso cliente.

Crie um arquivo com o nome de `FabricaFoguetesDirector.php`, conforme o código a seguir:

```php
<?php

namespace Builder;

class FabricaFoguetesDirector
{
    protected $construtorDeFoguetes;

    public function __construct(FogueteBuilder $construtorDeFoguetes)
    {
        $this->construtorDeFoguetes = $construtorDeFoguetes;
    }

    public function getFoguete(): FogueteProduct
    {
        return $this->construtorDeFoguetes->getFoguete();
    }

    public function construirFoguete()
    {
        $this->construtorDeFoguetes->buildModelo();
        $this->construtorDeFoguetes->buildMotores();
        $this->construtorDeFoguetes->buildTanqueCombustivel();
        $this->construtorDeFoguetes->buildNumeroLugares();
    }
}
```

Aqui fica claro o porquê da existência da classe abstrata `FogueteBuilder`, que serve como interface. Ela é a garantia para a classe `FabricaFoguetesDirector` de que todas as classes `Builder` conterão os métodos que precisamos. Na propriedade `$construtorDeFoguetes`, vamos armazenar uma instância de `FogueteBuilder` que corresponde aos nossos construtores concretos (**Concrete Builder**).

O que garante que a propriedade `$construtorDeFoguetes` possuirá somente valores com esse tipo de dado `FogueteBuilder` é o método construtor (`__construct`) dessa classe. Ele recebe uma variável como

parâmetro que só poderá ser deste tipo, conforme especificado na assinatura do método.

O método `construirFoguete` é o responsável por fazer a chamada aos métodos de montagem de nosso foguete. Ele chama os métodos definidos em nossa interface `FogueteBuilder` e insere as propriedades específicas de cada modelo de foguete, de acordo com qual construtor concreto (**Concrete Builder**) foi instanciado.

Por fim, temos o método `getFoguete`, que será chamado após a montagem do foguete. Ele chama o método `getFoguete` da interface `FogueteBuilder`, que no momento será uma de nossas classes de construtor concreto (**Concrete Builder**), e retorna a instância do nosso `FogueteProduct`.

Com todas as nossas classes implementadas, vamos agora fazer um teste de funcionamento. Crie um arquivo com o nome de `index.php` e insira o código a seguir:

```
<?php

require_once('../autoloader.php');

$montadoraDeFoguetesTestI = new Builder\FabricaFoguetesDirector(new
Builder\FogueteModeloIBuilder());
$montadoraDeFoguetesTestII = new Builder\FabricaFoguetesDirector(new
Builder\FogueteModeloIIBuilder());
```

Primeiro estamos instanciando a classe `FabricaFoguetesDirector`, passando para seu construtor uma instância de `FogueteModeloIBuilder` e atribuindo a variável `$montadoraDeFoguetesTestI`. Ela será responsável por montar um **Foguete Modelo I**.

Na linha seguinte, também instanciamos a classe `FabricaFoguetesDirector`, porém estamos passando para seu construtor uma instância de `FogueteModeloIIBuilder`. Isso quer dizer que o nosso foguete (`FogueteProduct`) será construído com as propriedades do **Foguete Modelo II**, o que muda é a nossa classe `Builder`.

Continuando, insira em seu arquivo as linhas a seguir:

```
$montadoraDeFoguetesTestI->construirFoguete();
echo $montadoraDeFoguetesTestI->getFoguete();

$montadoraDeFoguetesTestII->construirFoguete();
echo $montadoraDeFoguetesTestII->getFoguete();
```

Primeiro, chamamos o método `construirFoguete` que vai setar cada uma das propriedades de nosso **Foguete Modelo I**. Após isso, basta chamarmos o método `getFoguete` do nosso `Director` para obtermos o foguete montado.

Note que ele retorna uma instância de `FogueteProduct` , e que estamos tentando fazer um `print` do objeto pelo `echo` . É neste momento que o método mágico `__toString` é chamado. O mesmo processo é executado em nosso **Foguete Modelo II** pela variável `$montadoraDeFoguetesTestII` .

Se executarmos este código, a saída esperada será:

```
Foguete model: Foguete Modelo I
Litros do tanque: 1000
Número de motores: 3
Número de lugares: 8
Foguete model: Foguete Modelo II
Litros do tanque: 850
Número de motores: 2
Número de lugares: 6
```

Figura 5.3: Resultado esperado para o teste do Builder

A estrutura de arquivos ficou conforme a figura seguinte:

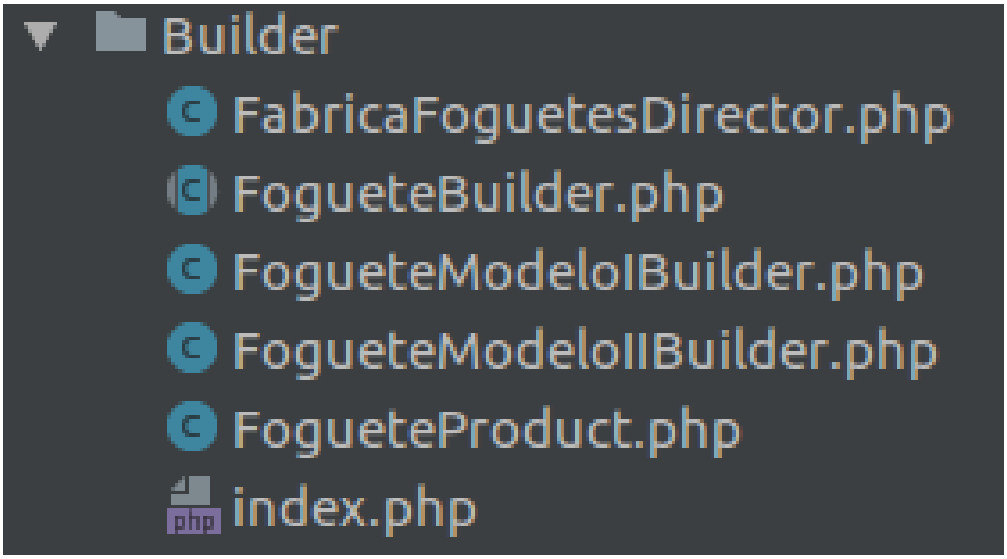

Figura 5.4: Estrutura de arquivos da implementação do Padrão de Projeto Builder

5.4 Conclusão

O Padrão de Projeto **Builder** é perfeito quando precisamos separar a construção de objetos complexos em pequenos passos, simplificando-os. Como desvantagem significativa de adotar este padrão, podemos destacar que, quando usamos em uma determinada classe de produto (**Product**), sempre teremos outros objetos para a construção de nossas instâncias, o que pode se tornar um problema em aplicações nas quais a performance é um ponto-chave.

Quanto às vantagens, temos a possibilidade de separar a definição das representações dos nossos produtos, o que facilita a manutenção e o desenvolvimento futuro. Também temos o código dessas representações e da construção totalmente encapsulado, e, como foi dito antes, temos um maior controle das etapas de criação de um objeto.

Padrões estruturais

Os padrões estruturais definem como classes e objetos são compostos entre si, permitindo alterações nessa composição em tempo de execução. Estão mais relacionados à estrutura como as classes e os objetos estão ligados.

Este livro aborda os seguintes padrões estruturais:

- Adapter;
- Facade;
- Decorator.

Capítulo 6
Tudo se encaixa com Adapter

Tipo: padrão estrutural

Nível de dificuldade:

6.1 O problema das diferentes interfaces

Imagine a seguinte situação: você trabalha como desenvolvedor mantendo um sistema de *e-commerce* (loja virtual). Este foi desenvolvido há 8 anos e utiliza um framework próprio que segue uma arquitetura MVC. Para quem não sabe o que significa uma arquitetura MVC, é uma das formas mais usadas atualmente para organização dos códigos de um sistema web, separando tudo em camadas de **M**odelo, **V**isão e **C**ontrole (daí a sigla **MVC**).

Simultaneamente a esse sistema, também é utilizado outro para gerenciamento de estoques dos produtos, preços, emissão de notas fiscais, entre uma série de outras coisas. Esse segundo é conhecido como um **ERP** e foi desenvolvido em outra linguagem, totalmente diferente do PHP.

No momento em que começaram a efetuar vendas na loja online, surgiu uma necessidade: era preciso informar ao **ERP** todas as ações feitas, como vendas, entradas de produtos, saídas etc. Caso contrário, poderiam ocorrer problemas graves no controle de estoque. Desde então, foi desenvolvido um módulo de integração para permitir essa comunicação entre o e-commerce e o ERP. Um módulo de integração nada mais é do que um conjunto de arquivos e classes que foram escritos em PHP e são responsáveis por notificar o ERP nos casos citados anteriormente.

Porém, de uns meses para cá, o dono do e-commerce decidiu que gostaria de reescrever o código todo do sistema com tecnologias mais novas, e passar a utilizar a nova versão do PHP para desenvolver um novo projeto, assim o desenvolvimento futuro se tornaria mais ágil, e a qualidade do projeto aumentaria. A estratégia seria reescrever todo o código base e, só depois, reescrever a parte de códigos do módulo de integração com o ERP, pois não havia tempo para reescrever o sistema completo no primeiro momento.

Aí chegamos ao problema: reescrever um projeto novo, com tecnologias mais novas e seguindo as melhores práticas de desenvolvimento, e continuar utilizando um módulo do sistema legado (sistema antigo). Isso não parece ser a melhor coisa a se fazer, pois esse módulo tem classes totalmente despadronizadas, desenvolvidas com metodologias de anos atrás, que nem são mais usadas. Entretanto, não podemos nos esquecer de que não há tempo para reescrever essa parte do sistema no momento.

Seria perfeito se tivéssemos a possibilidade de utilizar essa pequena parte (o módulo de integração) do sistema antigo no novo e, quando fosse reescrito, o sistema novo nem notaria a alteração.

Isto é, faríamos como se estivéssemos usando uma espécie de adaptador facilmente plugável que está entre duas partes do sistema.

6.2 O padrão Adapter

Definição

No mundo real, utilizamos adaptadores a todo momento. Um ótimo exemplo são os adaptadores elétricos de tomada: quando precisamos carregar um celular, não podemos conectar diretamente o cabo USB a uma tomada; precisamos de um adaptador que faça a conversão desses elementos incompatíveis.

Em um software, não temos tomadas ou cabos USB, mas temos classes que se comunicam entre si. Muitas vezes essas classes não conseguem se comunicar por possuir um conjunto de métodos e atributos diferentes. Para que ocorra essa comunicação, é necessário que alteremos uma das classes para que as duas consigam conversar.

Entretanto, nem sempre queremos alterar as classes do nosso sistema para isso, como visto no exemplo mostrado anteriormente. Com o uso do **Adapter**, é possível inserir um terceiro elemento que fica entre classes que queiram se comunicar. Este é um adaptador que faz essa comunicação entre interfaces incompatíveis.

Diagrama de classes

A seguir, veja o diagrama de classes do padrão **Adapter**:

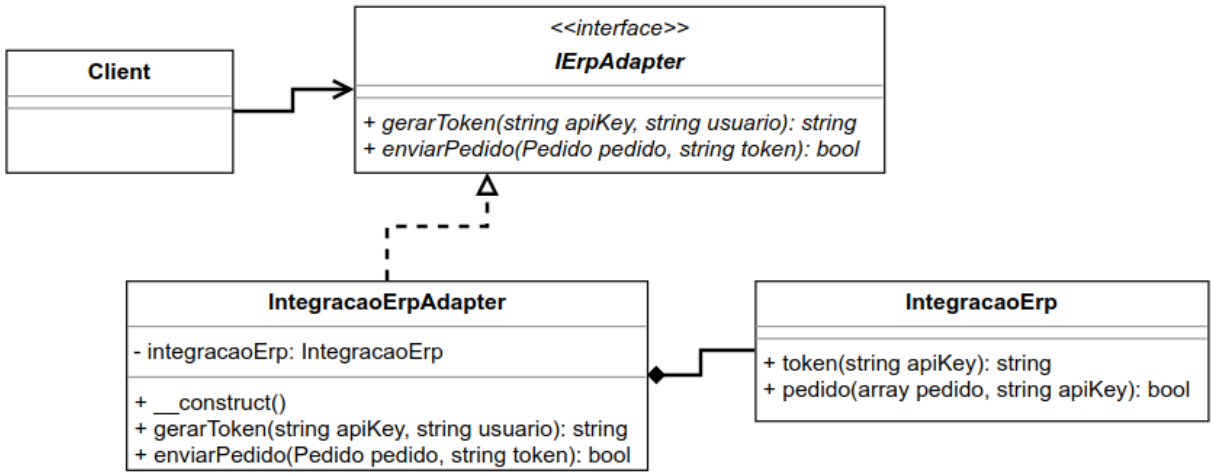

Figura 6.2: Diagrama de classes do Padrão de Projeto Adapter

Elementos que compõem o Adapter

- **Adaptee** (item adaptado): trata-se de uma interface já existente, da qual o nosso adaptador fará uso. Seria como a tomada da parede no exemplo.
- **Client** (cliente): são as classes que farão uso do adaptador. No exemplo do mundo real, o cliente seria o conector USB.
- **Target** (alvo): define a interface que o nosso cliente vai usar. Seria como o contrato que define que todos os conectores USB só funcionam com uma entrada USB (nada além disso).
- **Adapter** (adaptador): é o adaptador de fato, que implementa da interface de **Target**, porém, por trás, faz uso do item adaptado (**Adaptee**) de forma totalmente transparente. As classes que usam esse adaptador só têm conhecimento de sua interface, mas ele tem conhecimento da interface do item adaptado.

6.3 Resolvendo o problema do e-commerce

No problema apresentado neste capítulo, precisávamos de uma solução para conectar as duas interfaces diferentes de forma transparente. E como vimos, o **Adapter** encaixa-se perfeitamente

para essa solução. Agora veremos o quanto é simples a sua implementação com o PHP 7.

Vamos desenvolver um exemplo da ligação entre o novo e-commerce e o módulo do ERP (do sistema antigo). A seguir, temos a classe `IntegracaoErp`, que representa o elemento a ser adaptado (**Adaptee**) de nosso padrão. Ele seria o módulo de integração com o ERP, falado anteriormente.

Não queremos que as classes novas utilizem essa classe diretamente, nem que chamem seus métodos `token` e `pedido`, pois elas não estão de acordo com um padrão aceitável para o sistema novo. Por **padrão aceitável**, entenda possuir um código que evite complexidade desnecessária, e que já aplique funcionalidades disponibilizadas a partir da versão 7 do PHP (como tipagem de parâmetros, definição de tipo de retorno de métodos etc.).

```php
<?php

class IntegracaoErp
{
    public function token($apiKey)
    {
        // Código de solicitação do token aqui ...
        // ...

        return 'TOKEN_GERADO_VIA_API';
    }

    public function pedido($pedido, $apiKey)
    {
        // Código de envio do pedido aqui ...
        // ...

        return true;
    }
}
```

Geralmente, quando possuímos uma integração com outro sistema, para cada operação que vamos executar, precisamos gerar uma espécie de chave de permissão, conhecida como *token*. O método `token` representa o código necessário para fazer uma solicitação para a API do ERP, solicitando uma chave.

O parâmetro `$apiKey` é o identificador do sistema de origem que está requisitando um novo *token*, no nosso caso, o e-commerce. O uso dessa chave identificadora é um padrão usado em integrações entre sistemas.

Já o método `pedido` é o que representa a integração que informa ao ERP de cada novo pedido feito no e-commerce. O primeiro parâmetro, o `$pedido`, é um `array` com as informações básicas do pedido (preço, quantidade, código dos produtos); o segundo, o `$apiKey`, é o mesmo do método anterior, isto é, o identificador do sistema de origem.

Agora vamos à implementação da interface `IErpAdapter` que representa o **Target** do nosso padrão. Essa classe deve ficar no mesmo diretório da classe `IntegracaoErp`.

```
<?php

interface IErpAdapter
{
    public function gerarToken(string $apiKey, string $usuario): string;

    public function enviarPedido(Pedido $pedido, string $token): bool;
}
```

Como podemos perceber, ela possui exatamente a assinatura dos métodos que queremos para o novo sistema, utilizando as novidades do PHP 7 e com nomes de métodos mais claros. Porém, isso ainda não é o adaptador de fato; a interface foi criada para servir como contrato entre nosso cliente e nosso adaptador. Isso vai garantir que nosso adaptador terá todos os métodos que o nosso cliente conhece.

Os métodos `gerarToken` e `enviarPedido` são equivalentes a `token` e `pedido` da classe `IntegracaoErp`. No método `enviarPedido`, o primeiro parâmetro recebe um objeto do tipo `Pedido`. Ele não faz parte de nosso padrão, mas foi criado apenas para demonstrar as diferenças de tipos de dados dos parâmetros que podem ser passados para um adaptador, já que, neste método, recebemos as informações de um pedido dentro de um objeto.

Já no método `pedido` da classe `IntegracaoErp`, recebemos um `array`, ou qualquer outro tipo de dado que o desenvolvedor passe. Antes da versão 7 do PHP, não era possível definir o tipo de dado do parâmetro a ser passado – o que permitia passar qualquer tipo de coisa para o método e gerar erros não previstos.

Veja a implementação da classe `Pedido`:

```php
<?php

class Pedido
{
    protected $numeroPedido;

    protected $valorTotal;

    protected $produtos;

    public function addProduto(string $produto)
    {
       $this->produtos[] = $produto;
       return $this;
    }

    // Getter's e Setter's aqui ...
}
```

O funcionamento desta classe é bem simples. Ela é exatamente o que desenvolvedores chamam de **DTO** (*Data transfer object*) ou **VO** (**Value object**); serve apenas para transportar objetos com informações entre as camadas de seu sistema.

Cada instância de `Pedido` representa uma venda feita no sistema. Ele substitui a passagem de dados por arrays ou parâmetros diretamente, assim, temos um objeto com todas as informações que precisamos.

Até agora, vimos como ficou a construção da interface `IErpAdapter` que representa nosso contrato entre o cliente que usará o adaptador e a nossa classe que será adaptada (`IntegracaoErp`). Entretanto, isso não vai funcionar só com um contrato; precisamos do adaptador de fato. É ele que receberá as chamadas do cliente e, de forma transparente, adaptará essas chamadas para o item adaptado (*Adaptee*):

```php
<?php

class IntegracaoErpAdapter implements IErpAdapter
{
    private $integracaoErp;

    public function __construct(IntegracaoErp $integracaoErp)
    {
        $this->integracaoErp = $integracaoErp;
    }

    public function gerarToken(string $apiKey, string $usuario): string
    {
        return $this->integracaoErp->token($apiKey);
    }

    public function enviarPedido(Pedido $pedido, string $token): bool
    {
        $pedidoConvertidoArray = [
            'valor_total' => $pedido->getValorTotal(),
            'codigo' => $pedido->getNumeroPedido()
        ];

        foreach ($pedido->getProdutos() as $produto) {
            $pedidoConvertidoArray['produtos'][] = $produto;
        }
```

```
        return $this->integracaoErp->pedido($pedidoConvertidoArray,
$token);
    }
}
```

O primeiro ponto a ser observado é que a classe `IntegracaoErpAdapter` implementa a interface `IErpAdapter`, e isso obriga-a a escrever os métodos `gerarToken` e `enviarPedido`. Podemos notar que também existe um método adicional, o `__construct`, que é o construtor de classes no PHP (http://php.net/manual/en/language.oop5.decon.php).

Estamos utilizando duas boas práticas de programação, conforme explicado no primeiro capítulo: a injeção de dependência (pelo construtor) e a composição de objetos. Trata-se de uma composição, pois a classe `IntegracaoErp` está contida no adaptador (`IntegracaoErpAdapter`); o adaptador possui uma instância da classe adaptada, sem utilizar herança de classes. Com isso, concluímos que a classe `IntegracaoErpAdapter` faz uso de `IntegracaoErp`, no final das contas.

Conforme vimos anteriormente, o cliente só conhecerá a interface do adaptador, logo, ele somente usará os métodos `gerarToken` e `enviarPedido`. Ambos não possuem, de fato, a lógica responsável por suas ações. Por exemplo, o método `gerarToken` não gera um token diretamente; quem realmente continua gerando o token é a classe adaptada `IntegracaoErp`.

Entretanto, tudo isso ocorre de forma transparente para o cliente. Ele chama `gerarToken`, que repassa a solicitação para o método `token` da classe `IntegracaoErp`, na operação `$this->integracaoErp->token($apiKey)`.

O método `enviarPedido` funciona da mesma forma. Ele é a adaptação do método `pedido` da classe `IntegracaoErp`. No exemplo, estamos transformando o objeto `Pedido` em um `array`, que é o que o método `pedido` espera.

Uma consideração importante é que nem sempre teremos a necessidade de implementar a mesma quantidade de métodos (nem os mesmos parâmetros), em aplicações reais, por se tratar de interfaces diferentes. Um parâmetro pode ser passado para o adaptador (**Adapter**) sem ser usado inicialmente.

Por exemplo, poderíamos ter um método `enviarMensagem` na classe do adaptador e `enviarMensagem` na classe adaptada (este segundo é usado pelo primeiro). No método da classe adaptada, teríamos três parâmetros (`$mensagem` , `$remetente` e `$destinatario`) e, na classe adaptada, dois parâmetros (`$mensagem` e `$remetente`).

Enquanto usarmos a classe adaptada por trás do adaptador, o parâmetro `$destinatario` realmente não será utilizado, então, pareceria desnecessário este existir no adaptador. Porém, em um futuro quando a classe adaptada for substituída, poderíamos utilizar aquele parâmetro; bastaria alterar o adaptador para repassar isso para a classe adaptada.

Por fim, para finalizar o exemplo do Adapter, vamos implementar a seguir o código de nosso cliente (**Client**):

```
<?php

require_once('../autoloader.php');

$integracaoLegado = new \Adapter\IntegracaoErp();

$integracaoErpAdapter = new
\Adapter\IntegracaoErpAdapter($integracaoLegado);

$token = $integracaoErpAdapter->gerarToken('123456', 'GABRIEL_ANHAIA');

$pedidoDeTeste = new \Adapter\Pedido();
$pedidoDeTeste->setValorTotal(3509.50)
    ->setNumeroPedido(123456)
    ->addProduto('IPhone X')
    ->addProduto('Carregador Sem Fio - IPhone X');
```

```
$pedidoEnviadoComSucesso = $integracaoErpAdapter-
>enviarPedido($pedidoDeTeste, $token);

if ($pedidoEnviadoComSucesso) {
    echo 'Pedido envidado com sucesso!';
}
```

Repare que, nesse exemplo, não temos uma classe, então, colocaremos o código dentro de um arquivo `index.php`. Mas, em um projeto já em funcionamento, provavelmente esse código esteja dentro de seu projeto de uma forma mais estruturada; pode ser dentro de uma *Controller* ou qualquer outro tipo de classe, dependendo da arquitetura ou organização de seu sistema. O que importa é que esse arquivo refere-se ao cliente, e representa a parte do sistema que fará uso do Padrão de Projeto.

No código `"$integracaoLegado = new \Adapter\IntegracaoErp();"`, estamos criando uma instância do nosso módulo usado no sistema antigo. Uma pergunta que você pode estar se fazendo é: *"o que me impede de usar diretamente os métodos dessa classe?"*.

A resposta é que nada lhe impediria. Você pode tranquilamente chamar diretamente os métodos dela, porém estamos construindo o **Adapter** justamente para não fazer isso.

Depois, com `"new \Adapter\IntegracaoErpAdapter($integracaoLegado);"`, criamos a instância do adaptador e passamos para seu construtor a instância de `IntegracaoErp`. Aqui, entra a composição, quando `IntegracaoErpAdapter` recebe a instância de `IntegracaoErp`.

Na linha `$token = $integracaoErpAdapter->gerarToken('123456', 'GABRIEL_ANHAIA');`, já estamos fazendo uso do adaptador. Repare que chamamos o método `gerarToken`, que este está chamando o método `token` de `IntegracaoErp` e retornando para a variável `$token`.

Após isso, criamos uma instância de `Pedido` e definimos suas informações para enviar ao adaptador `$pedidoDeTeste = new \Adapter\Pedi...`. Essa instância do pedido e o token gerado são

enviados para o método `enviarPedido` do adaptador `$integracaoErpAdapter->enviarPedido($pedidoDeTeste, $token);`. O retorno é um `boolean` armazenado na variável `$pedidoEnviadoComSucesso`.

No final do código, verificamos se o pedido foi realmente enviado e exibimos uma mensagem de sucesso. Em nosso exemplo, o pedido sempre será enviado, por se tratar de um exemplo.

A estrutura de arquivos ficou conforme a figura seguinte:

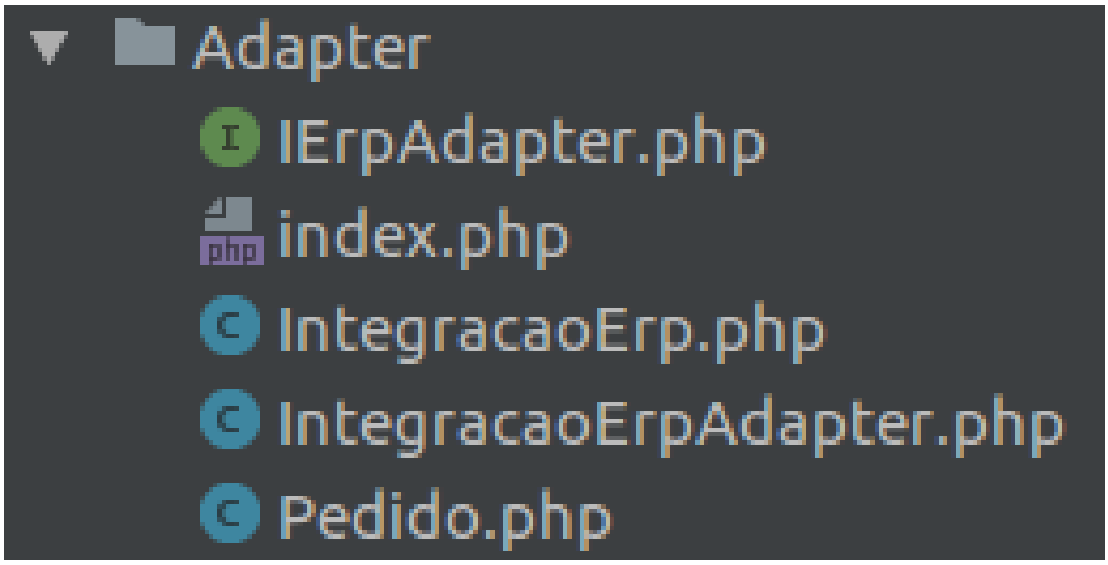

Figura 6.3: Estrutura de arquivos da implementação do Padrão de Projeto Adapter

ADAPTER DE OBJETO VERSUS ADAPTER DE CLASSE

Existem duas variações de implementação para o **Adapter**, sendo a primeira o de **Objeto** (usada no exemplo anterior). Desta forma, trabalhamos com a composição de objetos, o que é uma boa prática de programação segundo os princípios de SOLID, introduzidos por Robert C. Martin (Uncle Bob). Assim, cada adaptador (**Adapter**) possui uma instância de seu item a ser adaptado (**Adaptee**), conforme vimos detalhadamente no exemplo anterior.

Já o Adapter de **Classe** faz uso da herança. Cada classe de adaptador (**Adapter**) estende da classe a ser adaptada (**Adaptee**). Do ponto de vista funcional, temos o mesmo resultado no final, porém, diferente do adaptador de objeto, no qual podemos definir uma interface para a classe a ser adaptada, permitindo que um adaptador funcione com diferentes classes. Logo, cada classe terá seu próprio adaptador, que não será usado por nenhuma outra.

6.4 Conclusão

O maior benefício do **Adapter** é permitir que diferentes interfaces se comuniquem quando se tem a necessidade de conectar duas partes do sistema em que não devemos efetuar alterações; ou quando temos o objetivo de efetuar uma troca futura de partes do nosso sistema, sem que nosso cliente seja alterado.

Já vi muitos sistemas usarem esse padrão em algum momento. Sem dúvida, ele segue boas práticas de programação orientada a objetos e, quando bem empregado, pode trazer grandes benefícios, como a diminuição do acoplamento entre as classes e o

desenvolvimento de um sistema mais modular, dividindo um programa grande em vários menores.

CAPÍTULO 7
Descomplicando o sistema com Facade

TIPO: padrão estrutural

Nível de dificuldade:

7.1 O problema da alta complexidade dos subsistemas

Neste exemplo, vamos imaginar que você trabalha em uma fábrica de *software*, na qual são feitos projetos sob medida de acordo com a necessidade dos clientes.

Alguns meses atrás, você trabalhou em um projeto grande para uma biblioteca, no qual foram desenvolvidos muitos módulos internos no sistema (separados de acordo com as funcionalidades). Muitos desses módulos eram complexos de serem usados por outros desenvolvedores, pois exigiam diversas operações – entenda "diversas operações" como chamadas de métodos de classes distintas, passagem de muitos parâmetros, entre outras coisas.

Entre estes módulos, existe um para envio de e-mails, outro para controle de estoque de livros para locações, um para controle de clientes e um que se integra com uma **API** externa para envio de SMS. Uma **API** é um conjunto de padrões e rotinas para acesso de aplicações através da web.

Algumas semanas após a conclusão do projeto, outra empresa do mesmo segmento entrou em contato com a fábrica de software solicitando um sistema similar, porém, no lugar de diversas telas igual o projeto anterior, este teria poucas e executaria apenas algumas operações. Porém, necessitaria enviar e-mails, disparo de SMS e ter controle de locações – o que seria perfeito, pois isso tudo já tinha sido desenvolvido, bastaria importá-los para o sistema novo.

Mas temos um problema: esses módulos têm alta complexidade de uso. Para uma simples locação de um livro, diversos parâmetros e métodos devem ser chamados em uma sequência exata, e podemos precisar usar essas operações complexas em mais de um lugar pelo sistema. Isso viraria uma bagunça e se tornaria bem difícil de manter.

Seria ideal se pudéssemos abstrair todos esses passos necessários para seu uso e simplificá-los usando o mesmo código. É aí que entra o Padrão de Projeto **Facade**.

7.2 O padrão Facade

Definição

Facade (ou em português, "fachada") pode ser definido como uma fachada arquitetural de um sistema, ou seja, uma pequena parte do sistema que serve como camada para outras partes da aplicação. Quando precisamos usar esses subsistemas, trabalhamos diretamente em contato com a fachada.

Geralmente, essa camada de fachada é aplicada em situações nas quais possuímos um subsistema (sistema menor dentro de outro), ou módulos/bibliotecas que possuem uma certa complexidade para seu uso – seja por possuírem muitas funções, muitos parâmetros que devem ser passados para executar alguma ação ou, no pior dos

casos, por existir uma ordem exata na sequência de funções chamadas para executar uma determinada ação.

O **Facade** trabalha em um nível mais alto da aplicação e seu papel é simplificar o uso de módulos, subsistemas ou classes mais complexas. Mesmo adicionando essa camada sobre classes de nossa aplicação, ainda assim, se for necessário, podemos continuar usando classes diretamente. O **Facade** faz uso das outras classes sem a necessidade de alterações nelas.

Diagrama de classes

Veja o diagrama de classes do padrão **Facade**:

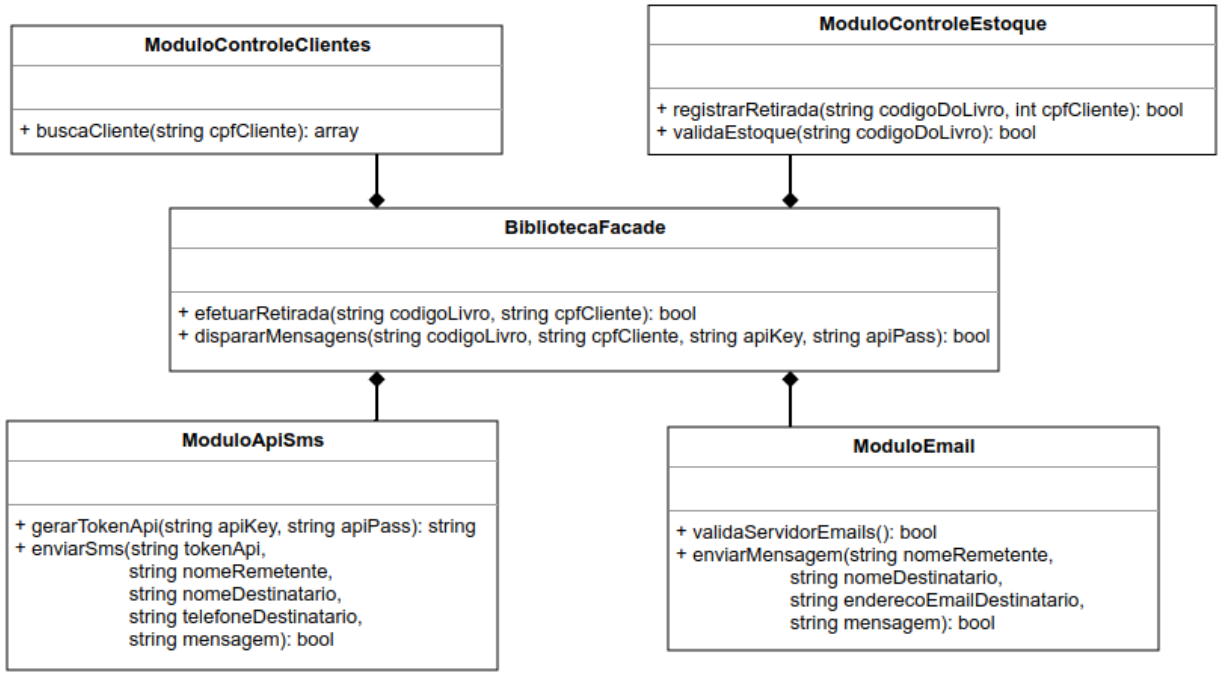

Figura 7.2: Diagrama de classes do Padrão de Projeto Facade

Elementos que compõem o Facade

- **Facade** (fachada): representação da fachada responsável por abstrair as classes de subsistemas. Geralmente, esta camada é construída sobre uma única classe.

- **Classes de subsistemas e módulos/bibliotecas**: são todas as classes das quais um **Facade** faz uso. São as classes cujas operações se quer simplificar.

7.3 Resolvendo o problema dos módulos altamente complexos

Para exemplificar o problema da alta complexidade de módulos e mostrar a dificuldade de seu uso, primeiro vamos iniciar desenvolvendo o código de cada um dos subsistemas citados no início deste capítulo.

Para organizar, crie uma pasta com o nome `Facade` para colocar os arquivos. O primeiro que desenvolveremos é um módulo para controle de clientes, que será usado posteriormente pela classe com a implementação do **Facade**. O nome do arquivo é `ModuloControleClientes.php` , e seu código:

```
<?php

namespace Facade;

class ModuloControleClientes
{
    public function buscaCliente(string $cpfCliente): array
    {
        // Simula a busca de um cliente pelo seu CPF.
        return [
            'nome' => 'Gabriel Anhaia',
            'telefone' => '99999999999',
            'email' => 'gabriel@mestredev.com.br'
        ];
    }
}
```

O objetivo da classe `ModuloControleClientes` é executar qualquer operação relacionada aos clientes do sistema. Neste exemplo, implementamos apenas o método `buscaCliente`. Como o próprio nome sugere, seu objetivo é buscar um cliente cadastrado no sistema, filtrando pelo seu CPF, que é passado por meio do parâmetro `$cpfCliente`. O retorno é um `array` com o cliente pesquisado.

Neste exemplo inserimos um `array` fixo a ser retornado; foi feito desta forma apenas para exemplificar. Então, precisamos de uma classe que representa um módulo de gerenciamento dos estoques dos livros da biblioteca e, para isso, vamos construí-la a seguir. O nome do arquivo é `ModuloControleEstoque.php`:

```
<?php

namespace Facade;

class ModuloControleEstoque
{
    public function registraRetirada(string $codigoDoLivro, int
$cpfDoCliente): bool
    {
        // Efetua o registro de retirada do livro.
        return true;
    }

    public function validaEstoque(string $codigoDoLivro): bool
    {
        // ... Valida se possui estoque
        echo "Registro de retirada efetuado com sucesso.\n";

        return true;
    }
}
```

Ela possui dois métodos: `validaEstoque` recebe no parâmetro `$codigoDoLivro` uma `string` com o código identificador do livro que se quer validar, e se há estoque disponível para retirada. Caso o livro

esteja disponível, a função vai retornar `true`. No nosso exemplo, sempre existirá estoque disponível.

O outro é `registraRetirada`, que serve para efetivar a retirada de um livro após o aluguel. Ele recebe como parâmetros o `$codigoDoLivro`, que é o identificador único do livro no sistema, e o `$cpfDoCliente`, que é o CPF do cliente a efetuar o aluguel. A função também retorna um `boolean` confirmando se a retirada foi concluída com sucesso – em nosso exemplo, sempre retornará `true`.

Agora precisamos criar uma classe que agrupe as funcionalidades relacionadas ao servidor de envio de e-mails do sistema. O próximo arquivo que vamos criar é o `ModuloEmail.php`:

```php
<?php

namespace Facade;

class ModuloEmail
{
    public function validaServidorDeEmails(): bool
    {
        // Valida se o servidor de emails está funcionando.
        return true;
    }

    public function enviaMensagem(
        string $nomeRemetente,
        string $nomeDestinatario,
        string $enderecoEmailDestinatario,
        string $mensagem
    ): bool {
        // Simula envio de mensagem de email.
        echo "Email enviado com sucesso.\n";

        return true;
    }
}
```

O método `validaServidorDeEmails` é usado para verificar se o servidor de e-mail está em funcionamento e disponível para o envio de mensagens. O retorno dessa função é um `boolean`, mas, no nosso exemplo, ele sempre retornará `true`.

O método `enviaMensagem`, como o nome sugere, serve para enviar e-mails aos clientes. Ele recebe quatro parâmetros: `$nomeRemetente`, o nome de quem está enviando a mensagem; `$nomeDestinatario`, o nome de quem se destina a mensagem; `$enderecoEmailDestinatario`, o endereço ao qual será enviada a mensagem; e `$mensagem`, o texto da mensagem que será enviada. Caso ela seja enviada com sucesso, a função retornará `true` (em nosso exemplo, isso sempre ocorrerá).

De acordo com o exemplo, temos três módulos, mas ainda precisamos desenvolver o código responsável pelo envio de mensagens para uma API que dispara por SMS para um determinado remetente. Crie um arquivo `ModuloApiSms.php`, conforme a seguir:

```
<?php

namespace Facade;

class ModuloApiSms
{
    public function geraTokenApi(string $apiKey, string $apiPass): string
    {
        // Simula a geração de um token através da API de envio de SMS.
        return 'TOKEN_RETORNADO_API';
    }

    public function enviaSms(
        string $tokenApi,
        string $nomeRemetente,
        string $nomeDestinatario,
        string $telefoneDestinatario,
        string $mensagem
    ): bool {
        // Simula envio de mensagem SMS via API.
```

```
        echo "SMS enviado com sucesso.\n";

        return true;
    }
}
```

O objetivo dessa classe é enviar uma mensagem para um sistema externo por meio de uma integração com sua API. Esse sistema externo ficará responsável por receber essa mensagem e enviar via SMS para o celular do destinatário.

Assim como a maioria das APIs necessitam de um token para seu uso, nosso exemplo não será diferente: para usar o método `enviaSms`, precisamos de um `token` de acesso que será gerado pelo método `geraTokenApi`. Este recebe dois parâmetros: `$apiKey`, que é o código identificador do nosso sistema, para que o sistema de envio de SMS saiba quem está solicitando o token; e `$apiPass`, que representa uma senha para acesso ao método de geração de token.

O segundo método `enviaSms` é responsável por efetivar o envio da mensagem que queremos disparar por SMS para alguém. O parâmetro `$tokenApi` é o token gerado anteriormente pelo método `geraTokenApi`, `$nomeRemetente` é o nome de quem está enviando a mensagem SMS, `$nomeDestinatario` é o nome da pessoa a quem se destina a mensagem, `$telefoneDestinatario` é o telefone que será usado para o destino da mensagem e, por fim, `$mensagem` é a mensagem de texto que será enviada para o telefone do destinatário.

Até agora, temos os quatro módulos necessários para nosso exemplo de **Facade**: `ModuloControleClientes`, `ModuloControleEstoque`, `ModuloEmail` e `ModuloApiSms`. Comentei anteriormente que o uso desses módulos possui uma certa complexidade. No exemplo posterior, demonstrarei o que seria necessário caso precisássemos utilizá-los em conjunto após a locação de um livro.

Não é necessário reproduzir esse código, apenas observe os pontos que serão comentados:

```php
<?php

$moduloControleEstoque = new ModuloControleEstoque();

$codigoLivro = '123456';
$cpfCliente = '12312312332';

if (!$moduloControleEstoque->validaEstoque($codigoLivro)) {
    throw new Exception('Estoque indisponível');
}

$moduloControleEstoque->registraRetirada($codigoLivro, $cpfCliente);

$moduloSms = new ModuloApiSms();
$moduloControleClientes = new ModuloControleClientes();
$moduloEmail = new ModuloEmail();

$cliente = $moduloControleClientes->buscaCliente($cpfCliente);

if ($moduloEmail->validaServidorDeEmails()) {
    $moduloEmail->enviaMensagem(
        'Biblioteca de Teste',
        $cliente['nome'],
        $cliente['email'],
        "Aluguel de livro de código '{$codigoLivro}' efetuado com
sucesso!"
    );
}

$token = $moduloSms->geraTokenApi($apiKey, $apiPass);

$moduloSms->enviaSms(
    $token,
    'Biblioteca de Teste',
    $cliente['nome'],
    $cliente['telefone'],
    "Aluguel de livro de código '{$codigoLivro}' efetuado com sucesso!"
);
```

No exemplo anterior, estão todos os passos necessários para a locação de um livro. Com esse código, estamos basicamente

executando as seguintes tarefas:

- Recebemos as informações com o código do livro e o CPF do cliente.
- Validamos se existe estoque do livro; caso exista, então efetivamos a retirada.
- Buscamos o cliente através de seu CPF.
- Validamos se o servidor de e-mails está funcionando.
- Caso esteja tudo certo, disparamos uma mensagem de e-mail informando sobre a confirmação da locação.
- Geramos um token para a API de envio de SMSs.
- Por último, enviamos a mensagem SMS.

Existe uma grande complexidade em executar essa operação, pois é preciso executar diversos passos. Além disso, também devemos respeitar uma ordem lógica. Imagine enviar primeiro a mensagem de confirmação, para depois confirmar que o livro não possuía estoque disponível? Isso não faria muito sentido.

No exemplo do problema da biblioteca, desenvolveríamos um sistema novo que faria uso de todos esses módulos. Para que o sistema novo não precise ter conhecimento de toda essa complexidade, implementaremos o **Facade**, que será a fachada acima disso. Vamos lá?

Crie um arquivo `BibliotecaFacade.php` com o seguinte código:

```
<?php

namespace Facade;

use Facade\{
    ModuloControleEstoque, ModuloControleClientes, ModuloEmail,
ModuloApiSms
};

class BibliotecaFacade
{
    public function efetuaRetirada(string $codigoLivro, string
```

```
$cpfCliente): bool
    {
        $moduloControleEstoque = new ModuloControleEstoque();

        if (!$moduloControleEstoque->validaEstoque($codigoLivro)) {
            throw new Exception('Estoque indisponível');
        }

        return $moduloControleEstoque->registraRetirada($codigoLivro,
$cpfCliente);
    }
}
```

A classe `BibliotecaFacade` vai receber as chamadas quando quisermos efetuar uma locação de livro. A primeira coisa que podemos notar de diferente é o comando `use` nas linhas iniciais:

```
use Facade\{
    ModuloControleEstoque, ModuloControleClientes, ModuloEmail,
ModuloApiSms
};
```

Como a nossa classe faz uso das demais, precisamos importá-las com o uso de namespaces, para que as classes sejam carregadas a partir do autoloader (explicado no primeiro capítulo). Essa sintaxe do exemplo está disponível a partir da versão 7 do PHP; anteriormente era necessário que, para cada classe, repetíssemos todo o caminho do namespace:

```
use Facade\ModuloControleEstoque;
use Facade\ModuloControleClientes;
...
```

Também é possível importar todas as classes dentro do namespace `Facade`, porém a única diferença perceptível seria algumas linhas a menos de código, nada que afetasse nossa aplicação. Optei por esta nova forma apenas para fins acadêmicos.

Até agora, temos um único método declarado na classe, o `efetuaRetirada`. Seu objetivo é diminuir toda a complexidade

necessária para efetuar uma simples locação de um livro. O método recebe dois parâmetros: o `$codigoLivro`, que é o identificador único do livro; e `$cpfCliente`, que representa o CPF do cliente que vai fazer a retirada.

No código `$moduloControleEstoque = new ModuloControleEstoque();`, estamos instanciando a classe `ModuloControleEstoque`. Depois, dentro do `if`, efetuamos a comparação `!$moduloControleEstoque->validaEstoque($codigoLivro)`. Caso não possua estoque disponível, lançamos uma exceção de erro; caso possua, chamamos a função `registraRetirada` do `ModuloControleEstoque` pela linha `$moduloControleEstoque->registraRetirada($codigoLivro, $cpfCliente);`. No final, retornamos um `boolean` confirmando se a locação teve sucesso.

Todas essas operações se resumem a apenas chamar o método `efetuaRetirada` do **Facade**. Observe o quão simples fica fazer a locação de um livro.

Agora vamos criar outra função dentro da classe `BibliotecaFacade`. Seu nome será `disparaMensagens` e ela ficará encarregada por abstrair toda a complexidade do envio de mensagens após a locação de um livro. Ela enviará mensagens por e-mail e SMS aos clientes, fazendo uso das classes `ModuloApiSms`, `ModuloEmail` e `ModuloControleClientes`.

```
public function disparaMensagens(
    string $codigoLivro,
    string $cpfCliente,
    string $apiKey,
    string $apiPass
): bool {
    $moduloSms = new ModuloApiSms();
    $moduloControleClientes = new ModuloControleClientes();
    $moduloEmail = new ModuloEmail();

    $cliente = $moduloControleClientes->buscaCliente($cpfCliente);
```

```
    if ($moduloEmail->validaServidorDeEmails()) {
        $moduloEmail->enviaMensagem(
            'Biblioteca de Teste',
            $cliente['nome'],
            $cliente['email'],
            "Aluguel de livro de código '{$codigoLivro}' efetuado com
sucesso!"
        );
    }

    $token = $moduloSms->geraTokenApi($apiKey, $apiPass);

    $moduloSms->enviaSms(
        $token,
        'Biblioteca de Teste',
        $cliente['nome'],
        $cliente['telefone'],
        "Aluguel de livro de código '{$codigoLivro}' efetuado com
sucesso!"
    );

    return true;
}
```

Nas primeiras linhas, instanciamos todas as classes que usaremos:

```
$moduloSms = new ModuloApiSms();
$moduloControleClientes = new ModuloControleClientes();
$moduloEmail = new ModuloEmail();
```

Para o envio da mensagem, precisamos de algumas informações do cliente, porém só temos seu CPF. Utilizamos o `ModuloControleClientes` para filtrá-lo:

```
$cliente = $moduloControleClientes->buscaCliente($cpfCliente);
```

Esse método `buscaCliente` da classe `ModuloControleClientes` retorna um `array` com as informações do cliente. Dentro do primeiro `if`, comparamos se o servidor de e-mails está funcionando (`$moduloEmail->validaServidorDeEmails()`); caso retorne `true`, chamamos a função `enviaMensagem` do módulo de e-mails e

passamos as informações de remetente, destinatário, e-mail e mensagem.

Então, precisamos enviar o SMS. O primeiro passo é gerar o token para permitir a comunicação com a API. Chamamos o método `geraTokenApi` do módulo de SMS, que nos retorna o token de acesso e é referenciado pela variável `$token`. Após isso, chamamos o método `enviaSms` do módulo de SMS e passamos o token gerado anteriormente junto às informações de remetente e destinatário. No final, retornamos `true` simbolizando o sucesso no envio das mensagens.

Novamente temos uma série de funções, instâncias de classes criadas e uma ordem que deve ser respeitada, porém, tudo isso está encapsulado dentro da função `disparaMensagens`. O sistema não precisa ter o conhecimento de toda a complexidade para enviar uma mensagem, apenas precisa conhecer essa função, que simplifica tudo.

Com a classe de **Facade** pronta, ainda assim precisamos de algo que chame suas funções, afinal, ela não se usará sozinha. Criaremos um arquivo `index.php` dentro da pasta `Facade` junto com os demais arquivos criados antes. Este arquivo simboliza a parte do nosso sistema que fará uso da nossa fachada (**Facade**):

```
<?php

require_once('../autoloader.php');

$bibliotecaFacade = new \Facade\BibliotecaFacade();

$codigoLivro = '123456';
$cpfCliente = '12345678998';

$efetuouRetirada = $bibliotecaFacade->efetuaRetirada($codigoLivro,
$cpfCliente);

if ($efetuouRetirada) {
    $bibliotecaFacade->disparaMensagens($codigoLivro, $cpfCliente,
```

```
'CHAVE_API', 'SENHA_API');
}
```

Note que não existe qualquer contato direto com os módulos. Não temos nenhum conhecimento de que eles existem, mas estamos instanciando a classe `BibliotecaFacade`, que faz uso de todos os módulos por detrás.

```
$bibliotecaFacade = new \Facade\BibliotecaFacade();

$codigoLivro = '123456';
$cpfCliente = '12345678998';
```

Na parte do código anterior, instanciamos essa classe e depois criamos um código de livro e um CPF de cliente, ambos fictícios.

```
$efetuouRetirada = $bibliotecaFacade->efetuaRetirada($codigoLivro,
$cpfCliente);
```

Nesse código, chamamos o método `efetuaRetirada` do **Facade**, que fica responsável por executar todas as operações complexas de forma transparente. O retorno da retirada do livro é um dado do tipo `boolean` e fica armazenado dentro da variável `$efetuouRetirada`.

Para finalizar a explicação do código, veja:

```
if ($efetuouRetirada) {
    $bibliotecaFacade->disparaMensagens($codigoLivro, $cpfCliente,
'CHAVE_API', 'SENHA_API');
}
```

Estamos verificando se a retirada foi efetuada com sucesso (no nosso exemplo, sempre retornará `true`), para depois chamarmos o método `disparaMensagens` do **Facade** para enviar as mensagens de e-mail e SMS para o cliente.

A estrutura de arquivos ficará conforme a figura seguinte:

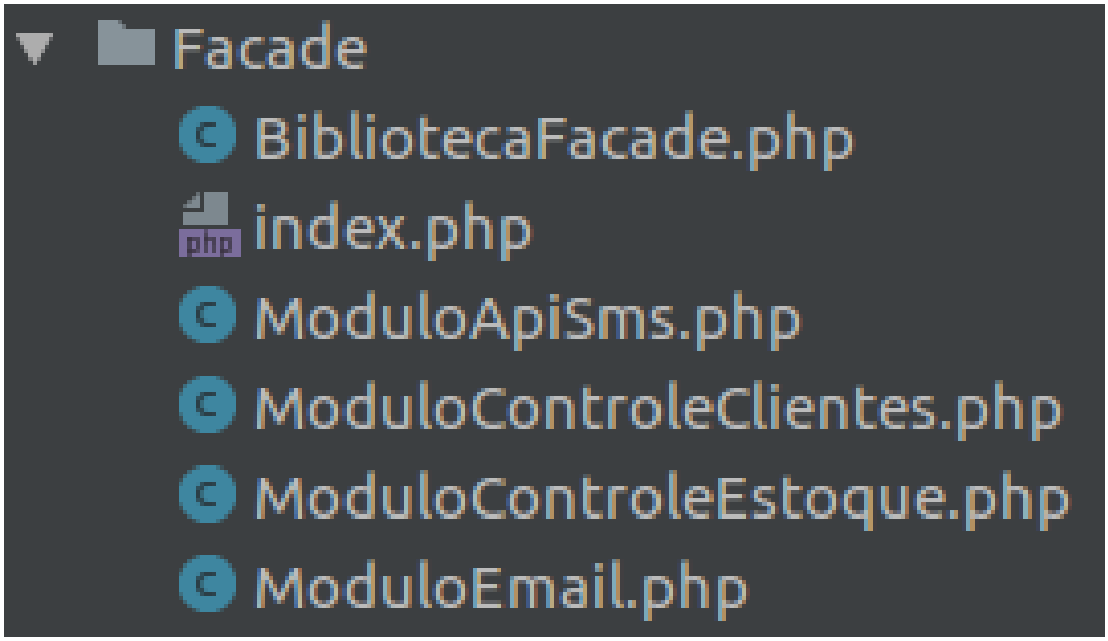

Figura 7.3: Estrutura de arquivos da implementação do Padrão de Projeto Facade

7.4 Conclusão

Utilizar o **Facade** é uma ótima escolha para os casos em que queremos simplificar ou abstrair uma parte do sistema, principalmente quando falamos de sistemas legados ou módulos que queremos utilizar indiretamente em nossa aplicação. O que torna-o ainda mais interessante é a possibilidade de continuar usando diretamente as classes, não afetando partes do sistema que já estão em funcionamento e serão alteradas.

CAPÍTULO 8
Agregando funcionalidades com Decorator

TIPO: padrão estrutural

Nível de dificuldade:

8.1 O problema dos pontos de ataque dos personagens

Imagine que você está desenvolvendo um jogo de MMORPG (*Massively Multiplayer Online RPG*, ou Jogo de RPG multijogador massivo online, em português), um daqueles jogos de RPG online que você joga diretamente no browser. Quando um novo jogador se cadastra, ele inicia a criação de seu personagem com três opções de especialidade: Mago, Cavaleiro ou Arqueiro.

Cada uma das especialidades possui sua própria pontuação de força de ataque, usada nas batalhas, que pode ser aprimorada à medida que o jogador vai evoluindo. Além disso, ele pode aumentar seu ataque com o uso de equipamentos especiais (espadas, acessórios, armaduras etc.).

Agora pense em como poderíamos desenvolver esta parte do sistema responsável por equipar os itens e aumentar a pontuação de ataque. Cada item tem seu próprio bônus, que pode ser equipado em qualquer quantidade bem como combinados, sem restrição na ordem em que os itens são equipados.

Por exemplo, um personagem com o seu ataque base igual a 10 pode equipar uma espada que aumenta 6 pontos, e depois dois anéis mágicos, que aumentam mais 3 pontos cada. No total, o personagem ficaria com 22 pontos de ataque. Perceba que a ordem em que os equipamentos são usados não importa. Porém, estes itens devem funcionar como uma espécie de agregação, que somará e aumentará a pontuação total.

Você pode pensar como solução desenvolver classes que já possuem a soma de todas as combinações possíveis de itens com as especialidades de personagens. Isso não seria viável para nosso problema, pois teríamos um número indeterminado de itens, e constantemente seriam lançados novos itens e novas especialidades de personagens.

Imagine o número de combinações possíveis: seriam infinitas classes que cresceriam cadas vez mais, tornando muito difícil a expansão e a manutenção. Com o Padrão de Projeto **Decorator**, podemos resolver este problema. Vejamos a seguir como fazer isso.

8.2 O padrão Decorator

Definição

O Padrão de Projeto **Decorator**, também conhecido como *Wrapper*, permite adicionar a um objeto comportamentos de forma dinâmica, agregando funcionalidades ou responsabilidades adicionais a ele. Ao contrário do uso da herança, em que são definidos comportamentos a classes, o Decorator permite adicionar esses comportamentos a objetos em tempo de execução, o que torna a aplicação muito mais dinâmica, permitindo mais flexibilidade a seus atributos e métodos.

Um exemplo muito usado para a explicação deste padrão é o de uma loja que vende bebidas e drinks. Temos as bebidas base, café expresso e *cappuccino*, e cada uma tem seu valor padrão a ser cobrado. Porém, é possível adicionar doses extra de alguns itens (dose de café, dose de leite, canela), sendo que é somado um valor ao total a cada item agregado. Note que os clientes podem adicionar doses de qualquer item adicional, independentemente da ordem.

Agora, imagine a implementação dessas bebidas com suas combinações. Poderíamos utilizar a herança e desenvolver muitas classes com todas as combinações possíveis. Entretanto, o número de classes seria um absurdo, sem falar da manutenção, inviável. Poderiam surgir mais bebidas e adicionais no futuro. Com o **Decorator**, temos uma estrutura de agregação de componentes que simplifica tudo isso.

A implementação é feita por meio de uma composição recursiva de objetos, que são os decoradores (que agregam as funcionalidades). Todos eles possuem a mesma interface e podem possuir uma instância de sua própria interface. Um decorador fica dentro de outro e, ao final, eles vão chamando os mesmos métodos de forma recursiva até chegar ao último elemento.

Diagrama de classes

A seguir, veja o diagrama de classes do padrão **Decorator**:

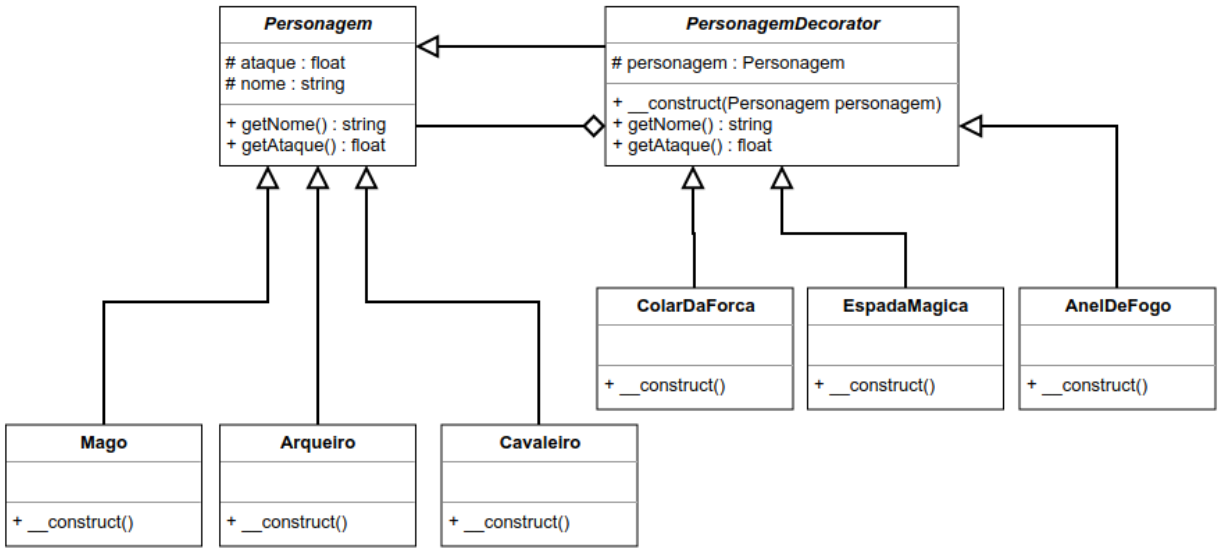

Figura 8.2: Diagrama de classes do Padrão de Projeto Decorator

Elementos que compõem o Decorator

- **Component** (componente): é a interface que será implementada, tanto pelos decoradores quanto pelos componentes concretos. Ela garantirá que ambas as classes tenham uma compatibilidade de métodos para o funcionamento das decorações.
- **Concrete Component** (componente concreto): é o componente do sistema que receberá as decorações. Nele, serão adicionados funcionalidades, comportamentos e estados extra. Estes componentes são considerados as classes base de nosso sistema, que serão alteradas.
- **Decorator** (decorador): é a interface ou classe abstrata que serve de contrato para os decoradores concretos implementarem, garantindo a compatibilidade para as decorações.
- **Concrete Decorator** (decorador concreto): são as implementações concretas dos decoradores. Elas contêm as funcionalidades extras a serem agregadas ao componente principal.

8.3 Resolvendo o problema dos personagens

Para demonstrar como podemos resolver o problema da pontuação dos personagens, primeiro vamos construir as classes base que receberão as decorações que contêm as funcionalidades extras. Em nosso exemplo, elas são as especialidades de personagens disponíveis no jogo: `Arqueiro`, `Cavaleiro` e `Mago`.

Como estas classes possuem características em comum, vamos criar uma classe abstrata base para todas, que servirá como interface para nossos decoradores e será representada pelo elemento `Component` de nosso padrão.

Crie uma pasta com o título de `Decorator`, e, dentro dela, crie um arquivo com o nome de `Personagem.php`. Siga o exemplo:

```
<?php

namespace Decorator;

abstract class Personagem
{
    protected $ataque;

    protected $nome;

    public function getNome(): string
    {
        return $this->nome;
    }

    public function getAtaque(): float
    {
        return $this->ataque;
    }
}
```

A classe `Personagem` foi definida como `abstract`, pois não poderá ser instanciada diretamente, apenas estendida pelas demais. Não faz

sentido para nosso sistema possuir um personagem sem uma especialidade de jogador (`Arqueiro` , `Mago` , `Cavaleiro`).

Nossa classe de personagem possui um atributo `$ataque` do tipo `protected` , que ficará armazenado na pontuação de ataque de nossos personagens e dos decoradores. O segundo atributo é o `$nome` , que será usado pelas classes base dos personagens (especialidades) para armazenar o identificador com seus nomes.

Um exemplo será a classe `Arqueiro` , que terá o valor do nome como `Arqueiro` . Será apenas um identificador fácil de ser recuperado pelo programa em tempo de execução.

Existe um método `getNome` , responsável por retornar a `string` com o nome da especialidade do personagem, armazenada na propriedade `$nome` . O método `getAtaque` é responsável por retornar um dado do tipo `float` com a pontuação de ataque do personagem, armazenada na propriedade `$ataque` .

Agora precisamos criar as classes base dos nossos personagens, as que de fato serão instanciadas em nosso sistema e representam as especialidades de jogador. Crie um arquivo com o nome de `Cavaleiro.php` e, dentro dele, siga o exemplo:

```
<?php

namespace Decorator;

class Cavaleiro extends Personagem
{
    public function __construct()
    {
        $this->nome = 'Cavaleiro';
        $this->ataque = 13;
    }
}
```

Na sequência, crie um arquivo com o nome de `Arqueiro.php` e insira o código a seguir:

```php
<?php

namespace Decorator;

class Arqueiro extends Personagem
{
    public function __construct()
    {
        $this->nome = 'Arqueiro';
        $this->ataque = 9;
    }
}
```

Para finalizar nossas classes base, crie um arquivo com o nome de `Mago.php` e, dentro, insira o código:

```php
<?php

namespace Decorator;

class Mago extends Personagem
{
    public function __construct()
    {
        $this->nome = 'Mago';
        $this->ataque = 10;
    }
}
```

As classes `Arqueiro`, `Cavaleiro` e `Mago` são muito semelhantes. Note que todas estendem os comportamentos da classe `Personagem`. Desta forma, garantimos que todos os nossos tipos de personagem possuirão os atributos `$ataque` e `$nome`, e também que ambas classes possuirão os métodos que encapsulam essas propriedades (`getNome` e `getAtaque`).

Apesar das semelhanças que podemos ver nos exemplos, existe uma diferença em seus métodos construtores (`__construct`). Iniciamos as propriedades com valores diferentes. O `Cavaleiro` inicia-se com o nome de sua respectiva especialidade, e a

pontuação de ataque de `13`; já o `Arqueiro` com um ataque de `9` e o `Mago` com `10`.

Com isso, já possuímos as classes base de nosso jogo e é possível criar personagens. O que ainda não temos claro é a forma como vamos adicionar pontos de ataque ao equipar itens, armas e acessórios ao nosso personagem.

Neste exemplo, cada equipamento do jogo será considerado como um decorador. Para que seja possível essa agregação de pontos pelos equipamentos, vamos iniciar construindo a classe base para nossos decoradores. Crie um arquivo com o nome de `PersonagemDecorator.php` e siga o exemplo:

```php
<?php

namespace Decorator;

abstract class PersonagemDecorator extends Personagem
{
    /** @var Personagem $personagem */
    protected $personagem;

    public function __construct(Personagem $personagem)
    {
        $this->personagem = $personagem;
    }

    public function getNome(): string
    {
        return $this->personagem->getNome();
    }

    public function getAtaque(): float
    {
        return $this->personagem->getAtaque() + $this->ataque;
    }
}
```

A classe `PersonagemDecorator` corresponde ao elemento `Decorator` do nosso padrão. Note que ela é abstrata que estende de outra classe abstrata. Isso parece incomum, mas é justamente o que queremos: assim como a `Personagem`, essa classe não será instanciada diretamente, já que seu objetivo é servir de base para todos os decoradores concretos.

Ela também herda o atributo `$ataque` da sua superclasse, pois nossos decoradores também possuirão um valor definido para essa propriedade. Este, no final, será somado para obtermos o total de ataque.

Esta classe possui um atributo `$personagem`, cuja visibilidade foi definida como `protected`, e ele só será acessado pela própria classe ou por classes-filhas. Quanto ao tipo de dado armazenado na variável, se estivéssemos aplicando o padrão em linguagens como **Java**, seu tipo seria `Personagem` (só aceitaria instâncias de `Personagem`), porém, ainda não é possível fazer isso em PHP devido à sua linguagem fracamente tipada.

Para que possamos ter uma maior garantia de que não será colocado nenhum outro tipo de dado, vamos centralizar o preenchimento dessa variável no método construtor da classe (`__construct`). Assim, evitamos que algum programador insira uma `string` ou quaisquer outros tipos de dados não esperados.

No método construtor, recebemos como parâmetro uma instância de `Personagem`, que é atribuída à propriedade `$personagem`. Se pararmos para analisar nossas classes, veremos que tanto os decoradores quanto nossas classes que representam as especializações (`Arqueiro`, `Mago` e `Cavaleiro`) são do tipo `Personagem`, e é justamente isso que queremos, pois poderemos adicionar uma funcionalidade a outra.

Note que sobrescrevemos os métodos `getNome` e `getAtaque`. Ambos já haviam sido implementados na classe `Personagem`, mas fizemos

isso porque são apenas as classes das especializações que buscam seu próprio valor neste método.

No método `getNome`, estamos buscando e retornando o nome da especialização do personagem que está dentro do decorador. Já no `getAtaque`, estamos buscando o ataque do personagem principal e somando ao ataque do decorador.

Pense que o decorador é uma espada que possui o ataque de 5; o resultado será o ataque do personagem principal a quem ela está decorando somado aos 5 pontos da própria espada. Porém, o método `getNome` retorna apenas o nome da especialidade do personagem, e foi definido assim pois a especialidade do personagem principal não muda à medida que adicionamos equipamentos novos.

Esse atributo não é afetado por nossos decoradores, é apenas sua pontuação de ataque que, no final, será somada com a pontuação dos decoradores. Assim, agora que implementamos a superclasse dos decoradores, vamos desenvolver as classes de exemplo dos decoradores específicos. Estes serão os equipamentos que vão elevar a pontuação de ataque dos nossos personagens.

Crie um arquivo com o nome de `EspadaMagica.php`. A seguir, observe o exemplo dessa classe:

```
<?php

namespace Decorator;

class EspadaMagica extends PersonagemDecorator
{
    public function __construct(Personagem $personagem)
    {
        parent::__construct($personagem);
        $this->ataque = 5;
    }
}
```

O funcionamento é bem simples: essa classe estende de nosso decorador base `PersonagemDecorator`. Em seu construtor (`__construct`), recebe uma instância de `Personagem`, que é repassada para o construtor de `PersonagemDecorator` pelo uso do `parent`. Depois, ele define o ataque referente ao item.

Neste caso, estamos dizendo que a Espada Mágica adiciona 5 pontos de ataque aos personagens que a usar. Vale ressaltar que a classe `EspadaMagica` também possui o atributo `$ataque` pela herança, isto é, ela também é uma classe do tipo `Personagem` por estender a `PersonagemDecorator`.

Para que nosso exemplo fique mais claro, vamos desenvolver mais dois decoradores de equipamentos para os personagens. Crie um arquivo com o nome de `AnelDeFogo.php` e insira o seguinte código:

```
<?php

namespace Decorator;

class AnelDeFogo extends PersonagemDecorator
{
    public function __construct(Personagem $personagem)
    {
        parent::__construct($personagem);
        $this->ataque = 3;
    }
}
```

Agora vamos criar o último decorador de exemplo. Crie um arquivo com o nome de `ColarDaForca.php`; o código deve ficar conforme a seguir:

```
<?php

namespace Decorator;

class ColarDaForca extends PersonagemDecorator
{
    public function __construct(Personagem $personagem)
```

```
    {
        parent::__construct($personagem);
        $this->ataque = 2;
    }
}
```

As classes `AnelDeFogo` e `ColarDaForca` são muito semelhantes à `EspadaMagica`. A única diferença é que a pontuação de ataque definida em seus construtores muda, já que cada item tem seu próprio benefício.

Com isso, concluímos as classes que fazem parte do nosso padrão. Vamos criar um arquivo com o nome de `index.php` para que possamos testá-lo na prática e entender um pouco mais como ele funciona. Dentro deste arquivo, insira as seguintes linhas de código:

```
<?php

require_once('../autoloader.php');

$decorator = new \Decorator\Mago();
```

Estamos instanciando a nossa classe `Mago`. Lembre-se de que ela é considerada uma classe base, pois representa uma especialidade do personagem.

A seguir, adicione as seguintes linhas:

```
$decorator = new \Decorator\AnelDeFogo($decorator);
$decorator = new \Decorator\AnelDeFogo($decorator);
$decorator = new \Decorator\ColarDaForca($decorator);
$decorator = new \Decorator\EspadaMagica($decorator);
```

Na primeira linha, estamos instanciando o nosso primeiro decorador, o `AnelDeFogo`. Note que passamos para seu construtor uma instância da nossa classe base `Mago`. Na linha seguinte, criamos outro decorador de `AnelDeFogo` e passamos para seu construtor o decorador anterior. Na linha seguinte, instanciamos a classe `ColarDaForca` repetindo o processo, e, por último, a classe `EspadaMagica`.

O que muitas pessoas pensam quando ouvem falar sobre este padrão é que possuímos uma classe principal (no exemplo, as especialidades) composta por outras classes que a decoram (no nosso exemplo, os equipamentos). Porém, o que realmente acontece é que estamos adicionando uma classe dentro da outra, como uma espécie de pilha.

Esta pilha de objetos pode ser observada dentro da nossa variável `$decorator`. Na última linha do exemplo, ela possui uma instância de `EspadaMagica`, na qual há uma instância de `ColarDaForca`. Dentro desta, há uma instância de `AnelDeFogo`, onde existe outra instância de `AnelDeFogo`, que, por fim, contém uma instância da nossa classe base, `Mago`.

A organização fica da seguinte forma: **EspadaMagica > ColarDaForca > AnelDeFogo > AnelDeFogo > Mago**. Porém, poderíamos continuar adicionando infinitos decoradores em qualquer ordem.

Agora, vamos adicionar o código responsável por buscar o identificador (nome) da especialização do personagem:

```
echo $decorator->getNome();
echo "\n";
```

O que acontece aqui é que o método `getNome` vai ser chamado em cada uma das instâncias de decoradores, e a ordem é a mesma mostrada anteriormente: **EspadaMagica > ColarDaForca > AnelDeFogo > AnelDeFogo > Mago**. Uma a uma, as instâncias vão chamando o método `getNome` e seguindo a hierarquia.

Este método está definido dentro da classe `PersonagemDecorator` e é executado em `return $this->personagem->getNome()`. Em cada um dos decoradores, a propriedade `$this->personagem` é o seu decorador seguinte – exceto no último, que chama a classe base `Mago`. Esta classe possui uma implementação diferente de `getNome`, pois retorna seu próprio nome com o código `return $this->nome;`. Assim, no final, temos o nome da especialidade do personagem.

O comando `echo "\n";` serve para a quebra de linha em nossa saída de texto. Para finalizar nosso exemplo, vamos inserir o código responsável por buscar e exibir a pontuação de ataque do personagem:

```
echo "Ataque = {$decorator->getAtaque()}";
echo "\n";
```

O método `getAtaque` segue a mesma ideia do anterior, que é chamado recursivamente por meio de nossos decoradores. No exemplo, chamamos o método `getAtaque` do decorador `EspadaMagica`, que define 5 pontos de ataque.

Ele retorna esse valor somado à pontuação de ataque do próximo resultado de `getAtaque`, da hierarquia de decoradores. Por fim, ele chega à classe base do nosso personagem, e retorna sua pontuação padrão de ataque. No final, temos a soma completa da pontuação.

1. `EspadaMagica` = 5 pontos
2. `ColarDaForca` = 2 pontos
3. `AnelDeFogo` = 3 pontos
4. `AnelDeFogo` = 3 pontos
5. `Mago` = 10 pontos

Por fim, teremos o resultado `23`. Esta é a pontuação de ataque de nosso personagem quando somada aos valores de cada decorador. Ao executarmos este arquivo, temos a seguinte saída:

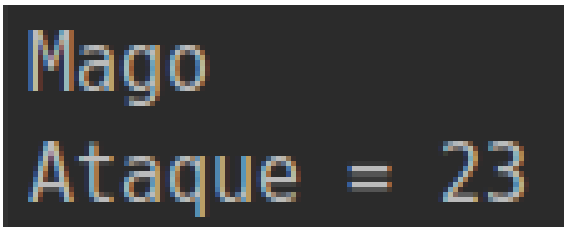

Figura 8.3: Resultado esperado para o teste do Decorator.

Com isso, concluímos a implementação do nosso exemplo. No final, temos a nossa classe base `Mago`, na qual são adicionadas

funcionalidades (pontuação de ataque). Em outras implementações, poderiam ser adicionadas funcionalidades como comportamentos diferentes.

Há uma consideração a ser feita a respeito da implementação do padrão com a classe `Personagem` como uma classe abstrata. Quem já leu o livro do GOF sobre Padrões de Projetos percebeu que o exemplo deles foi construído como uma **interface**. Ambas as soluções estão corretas, porém, no meu caso, mostrei os decoradores trabalhando com atributos (ou seja, `$ataque`), então, fez mais sentido utilizar uma classe abstrata.

A estrutura dos arquivos ficou conforme a figura seguinte:

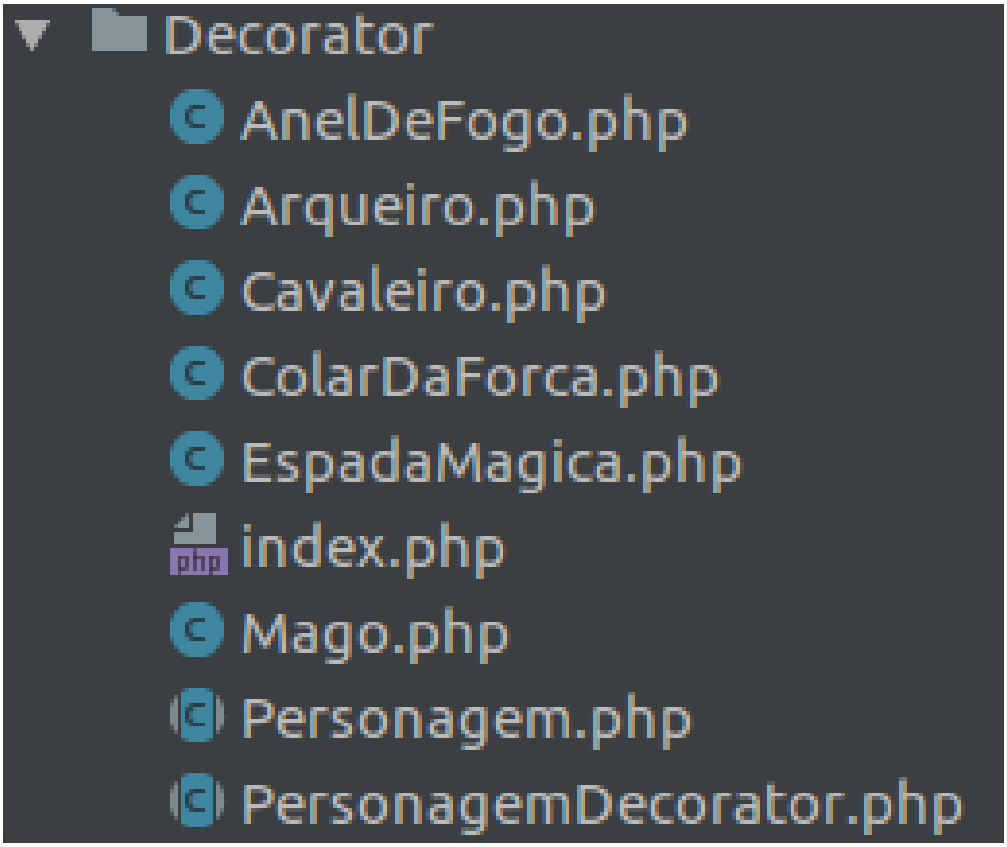

Figura 8.4: Estrutura de arquivos da implementação do Padrão de Projeto Decorator

LINGUAGEM TIPADA

Linguagens tipadas são linguagens de programação que usam variáveis com tipos específicos de dados. As *fortemente tipadas* são aquelas em que definimos os tipos de variáveis no momento de sua criação. Alguns exemplos disso são **Java** e **C#**.

Já nas linguagens fracamente tipadas, como o PHP, não existe definição de tipo. O desenvolvedor pode criar uma variável que armazena uma `string` e, logo em seguida, atribuir um `float` a ela. Isso não geraria erros de execução.

8.4 Conclusão

Podemos observar algumas desvantagens que podem surgir ao utilizar esse padrão. Sem dúvida, o padrão pode gerar um número de classes muito grande em nosso projeto, o que pode, em vez de facilitar, acabar dificultando sua leitura e manutenção.

Porém, existem diversas vantagens. Diferente da herança, que adiciona funcionalidades a toda uma cadeia de classes sem muita flexibilidade, o **Decorator** permite adicionar funcionalidades a objetos dinamicamente e em tempo de execução.

Essas funcionalidades são adicionadas individualmente a um objeto e podem ser pré-selecionadas. Deste modo, evitamos classes cheias de métodos que nunca serão usados, bem como alterações no código para novas combinações de objetos com funcionalidades variadas.

Uma outra grande vantagem do padrão é que as classes de componentes se mantêm simples, apenas sendo necessário desenvolver novos decoradores caso se queira uma nova

funcionalidade. A construção de decoradores, quando feita de forma padronizada, torna o código muito mais fácil de ler e expandir.

Padrões comportamentais

Definem a comunicação entre classes e objetos, e também seus comportamentos. Está ligado diretamente à atribuição de responsabilidades desses objetos e como os algoritmos se comportam.

Este livro aborda os seguintes padrões:

- Memento;
- Observer;
- Template Method;
- Strategy.

CAPÍTULO 9
Capturando o estado atual de um objeto com Memento

TIPO: padrão comportamental

Nível de dificuldade:

9.1 O problema do software de edição de texto

Você recentemente trabalhou em um projeto de desenvolvimento de um software de edição de texto. Quando você finalizou, ele passou por diversos roteiros de homologação para garantir que tudo funcionaria conforme o previsto e ele pudesse ser liberado para a comercialização.

Estava tudo correndo na mais perfeita ordem, até que alguém, responsável pela homologação, identificou a falta de uma funcionalidade essencial, e foi aí que começou sua dor de cabeça. A funcionalidade identificada era de guardar versões anteriores de documentos editados através do programa, assim você poderia retornar para um estado anterior do documento que estivesse editando.

Assim como qualquer programa, existem milhares de formas de construir o código e obter quase que o mesmo resultado no final, porém o padrão **Memento** é perfeito para a necessidade de salvar e recuperar estados de um determinado objeto. Por meio dele, podemos resolver o problema destes estados facilmente.

9.2 O padrão Memento

Definição

O **Memento** é uma excelente alternativa para captura e restauração de estados internos de objetos. Seriam como backups de informações de um objeto: o padrão faz cópias destes estados e as mantém armazenadas em um local.

As informações a serem salvas dependem da necessidade de cada sistema. Por exemplo, em um jogo de videogame, os estados poderiam ser gravações (*saved game*) das fases alcançadas de um jogo.

A grande vantagem de utilizar esse padrão para isso é que ele não viola o encapsulamento dessas classes cujo estado se quer manter salvo. Além disso, sua utilização é muito simples e organizada.

Diagrama de classes

A seguir, veja o diagrama de classes do padrão **Memento**:

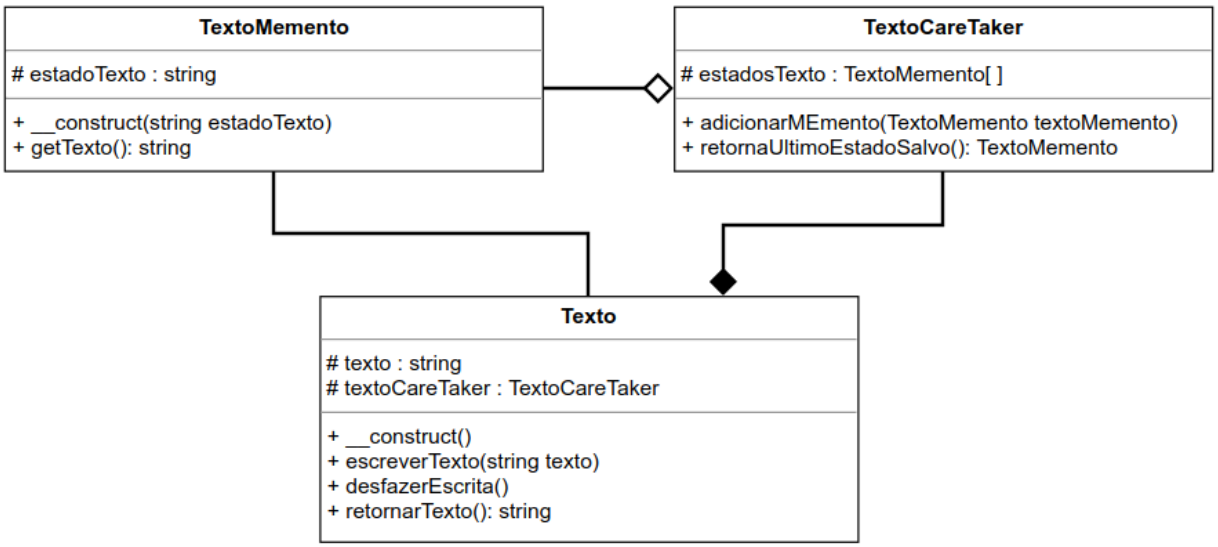

Figura 9.2: Diagrama de classes do padrão Memento em PHP

Elementos que compõem o Memento

- **Originator**: é o objeto cujo estado queremos capturar.
- **Memento**: é o objeto usado para armazenar o estado interno do `Originator`.
- **Caretaker** (armazenador): responsável por armazenar e gerenciar os estados internos (`Memento`) de um `Originator`.

9.3 Resolvendo o problema do editor de texto

Vamos aplicar o padrão **Memento** para resolver o problema do editor de texto. Para que possamos armazenar nossos estados do texto, precisamos de um local para isso. Então, a primeira parte que desenvolveremos é a classe responsável por armazenar os estados de todas as alterações de nosso objeto principal (editor de texto).

Crie uma pasta com o nome de `Memento` e, dentro dela, crie um arquivo `TextoMemento.php`. A seguir, está a implementação dessa classe:

```php
<?php

namespace Memento;

class TextoMemento
{
    protected $estadoTexto;

    public function __construct(string $estadoTexto)
    {
        $this->estadoTexto = $estadoTexto;
    }

    public function getTexto(): string
    {
        return $this->estadoTexto;
    }
}
```

A classe `TextoMemento` possui uma propriedade `$estadoTexto` com visibilidade `protected`, assim não poderá ser acessada diretamente de fora da classe, o que garante o encapsulamento dos dados. O objetivo desta propriedade é armazenar um estado de texto.

Temos o método construtor `__construct`, que recebe o estado de texto. Note que ele é obrigatoriamente do tipo `string`, conforme definido antes do parâmetro. Como a propriedade do estado é do tipo `protected`, necessitamos de uma forma para resgatar esse valor, e é por isso que implementamos o método `getTexto`, que retorna uma string com o estado salvo pela linha `"return $this->estadoTexto;"`.

Além da classe responsável por armazenar cada um dos estados, precisamos de um local centralizado onde ficarão todos os estados agrupados. Crie um arquivo `TextoCareTaker.php` e siga a implementação:

```php
<?php

namespace Memento;
```

```
class TextoCareTaker
{
    protected $estadosTexto = [];

    public function adicionarMemento(TextoMemento $textoMemento) {
        $this->estadosTexto[] = $textoMemento;
    }

    public function retornarUltimoEstadoSalvo(): TextoMemento
    {
        if (empty($this->estadosTexto)) {
            return null;
        }

        $textoMemento = end($this->estadosTexto);
        unset($this->estadosTexto[sizeof($this->estadosTexto) - 1]);

        return $textoMemento;
    }
}
```

Agora, a classe `TextoCareTaker` mantém todos os estados salvos agrupados, e centralizará o acesso a eles de forma organizada e controlada. A propriedade `$estadosTexto` é responsável por armazenar um `array` de instâncias de `TextoMemento`, e o que garantirá que todos os dados dessa lista sejam deste tipo será o método `adicionarMemento`. Somente ele acessa a propriedade da listagem para a adição de novos itens, conforme pode ser visto a seguir:

```
public function adicionarMemento(TextoMemento $textoMemento) {
    $this->estadosTexto[] = $textoMemento;
}
```

Ele recebe um objeto do tipo `TextoMemento` e incrementa no final do `array` de estados. Também temos o método responsável por retornar os estados, o `retornarUltimoEstadoSalvo`, que retorna um objeto do tipo `TextoMemento`; ou `null`, caso não exista mais nenhum estado a ser retornado.

No bloco de código a seguir, ele valida se a lista de estados está vazia; caso esteja, é retornado `null`:

```
if (empty($this->estadosTexto)) {
    return null;
}
```

Depois, buscamos o último estado adicionado na listagem e o apagamos:

```
$textoMemento = end($this->estadosTexto);

unset($this->estadosTexto[sizeof($this->estadosTexto) - 1]);
```

Seguimos o conceito de fila do tipo **LIFO** (*Last in, first out*): o último elemento a entrar na pilha é o primeiro a sair. Escolhemos utilizar esta forma em nossa ordem de recuperação de estados de texto para seguir o padrão da maioria de editores, que sempre retorna para o último estado salvo. No final do método, retornamos o último estado salvo na listagem.

Além da classe responsável por armazenar um estado específico e da que agrupa os estados, precisamos da classe que de fato possui o estado principal, o elemento `Originator` de nosso padrão. Crie um arquivo `Texto.php` e siga o exemplo:

```
<?php

namespace Memento;

class Texto
{
    protected $texto;

    protected $textoCareTaker;

    public function __construct()
    {
        $this->textoCareTaker = new TextoCareTaker();
        $this->texto = '';
```

```
    }

    public function escreverTexto(string $texto)
    {
        $this->textoCareTaker->adicionarMemento(new TextoMemento($this-
>texto));
        $this->texto .= $texto;
    }

    public function desfazerEscrita()
    {
        $this->texto = $this
            ->textoCareTaker
            ->retornarUltimoEstadoSalvo()
            ->getTexto();
    }

    public function retornarTexto(): string
    {
       return $this->texto;
    }
}
```

A classe `Texto` possui a propriedade `$texto`, na qual é armazenado o texto atual do editor. A propriedade `$textoCareTaker` armazena uma instância de `TextoCareTaker`, que agrupará os estados salvos de `$texto`.

No método construtor `__construct` dessa classe, inicializamos as duas propriedades: `$texto` com uma string vazia, e `$textoCareTaker` com a instância da classe `TextoCareTaker`. Quanto ao método `escreverTexto`:

```
public function escreverTexto(string $texto)
{
    $this->textoCareTaker->adicionarMemento(new TextoMemento($this-
>texto));
    $this->texto .= $texto;
}
```

Ele é chamado toda vez que escrevemos algo novo no texto, na linha:

```
$this->textoCareTaker->adicionarMemento(new TextoMemento($this->texto));
```

Estamos criando uma nova instância do objeto `TextoMemento` ao passar o parâmetro `$this->text`, que é o texto atual armazenado. Assim, a cada nova alteração do documento, teremos o estado anterior do texto salvo.

Esse estado, representado pelo objeto `TextoMemento`, é passado ao método `adicionarMemento` de `TextoCareTaker`, que o adiciona no fim da listagem de estados. Na linha seguinte, `$this->texto .= $texto;`, incrementamos o novo texto adicionado à propriedade `$texto`.

Quando precisarmos resgatar uma versão anterior do documento, chamaremos o método `desfazerEscrita`:

```
public function desfazerEscrita()
{
    $this->texto = $this
        ->textoCareTaker
        ->retornarUltimoEstadoSalvo()
        ->getTexto();
}
```

Estamos dizendo que `$this->texto` vai receber a versão anterior do texto salvo. Esse texto é retornado do objeto `$this->textoCareTaker`. Chamamos a função `retornarUltimoEstadoSalvo`, que retorna um objeto do tipo `TextoMemento`, para depois chamarmos `getTexto`, que retorna a `string` com o texto em seu estado anterior. Note que o estado anterior passa a ser seu estado atual.

O último método é o `retornarTexto`:

```
public function retornarTexto(): string
{
   return $this->texto;
}
```

Como o próprio nome sugere, ele serve para retornar a `string` de texto atual armazenada no objeto `Texto`. Para efetuar um teste dessas classes, crie um arquivo `index.php` e insira o código a seguir:

```
<?php

require_once('../autoloader.php');

$texto = new \Memento\Texto();

$texto->escreverTexto("A");
$texto->escreverTexto("B");
$texto->escreverTexto("C");

echo $texto->retornarTexto();
echo "\n";

$texto->desfazerEscrita();
echo $texto->retornarTexto();
```

Instanciamos um novo objeto de texto na linha `$texto = new \Memento\Texto();`. Depois, chamamos a função `escreverTexto` três vezes adicionando algumas letras. Na primeira chamada à função `retornarTexto`, a saída esperada é: `ABC`.

Após isso, chamamos a função `desfazerEscrita`, que retorna para o último estado antes da adição da letra `C`. A saída esperada da segunda chamada de `retornarTexto` deve ser: `AB`.

A estrutura dos arquivos ficou conforme a figura seguinte:

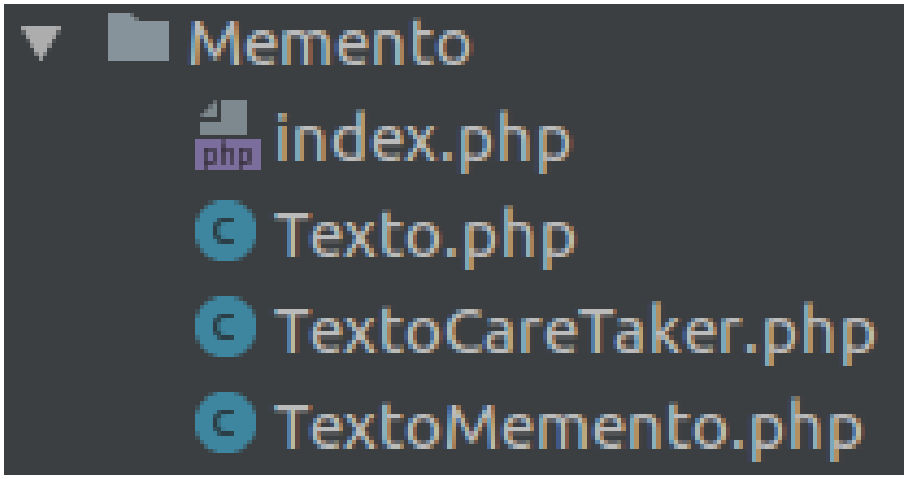

Figura 9.3: Estrutura de arquivos da implementação do Padrão de Projeto Memento

9.4 Conclusão

O Padrão de Projeto **Memento** torna-se uma ótima alternativa para gravação e recuperação de estados internos de objetos. Além de ter uma implementação extremamente simples, temos uma diminuição das responsabilidades de acesso aos estados do objeto `Originator` pela forma como o padrão se compõe.

A única desvantagem que podemos observar nele é que ele pode utilizar muita memória de um computador, caso não se tenha controle ou não se limite o número de instâncias ou dados salvos de um determinado estado. Um ponto importante é que somente o objeto originador dos estados (`Originator`) deve ter acesso a seus estados, assim temos um maior controle do acesso aos estados armazenados.

Capítulo 10
Atualização em tempo real com Observer

Tipo: padrão comportamental

Nível de dificuldade:

10.1 A confusão da baixa de estoque

Você trabalha como programador em um sistema *ERP*, desenvolvido em PHP 7+, que funciona através de qualquer navegador web, sem a necessidade de instalações de versões (como a maioria dos ERPs em funcionamento no mercado). Esse sistema é de uma empresa que vende aparelhos eletrônicos.

Ela vende muitos produtos diariamente, e necessita que seu controle de estoque esteja sempre atualizado e alinhado com o estoque físico disponível. Existe uma pessoa responsável por fazer a baixa de cada item no estoque de forma manual, por meio de uma tela do sistema. Cada vez que um estoque chega a zero, uma mensagem deve ser enviada por e-mail para o setor de compras da empresa, responsável por adquirir mais itens daquele produto.

Além de simplesmente enviar a mensagem, esse produto (seu identificador) deve ser enviado para uma listagem de produtos que devem ser comprados futuramente. Isso ocorre para que se tenha um maior controle dos itens com falta de estoque.

Imagine o envio de e-mails, o controle de requisições de compra e o controle de estoque como três classes distintas. As classes de e-mail e de controle de requisições de compra executam uma ação quando a classe de controle de estoques faz uma alteração na quantidade de um item. Poderíamos simplesmente instanciar as duas classes dentro da classe de estoque e, quando o produto chegasse a zero, bastaria chamar alguma função responsável por cada ação (e-mail, requisição de compra).

Sem dúvidas, este parece o mais simples a se fazer, porém teríamos um grande problema em desenvolver o sistema dessa forma. As três classes estariam altamente acopladas. Caso fosse necessário efetuar alguma alteração na classe de e-mails ou na de requisição de compras, teríamos problemas se não atualizássemos a classe de estoque, e isso poderia ocorrer para todas as outras classes do sistema que fazem uso delas. É por isso que devemos ter um acoplamento mais baixo entre classes dependentes umas das outras.

Além de buscarmos um baixo acoplamento entre essas classes do sistema, se repararmos no modelo do exemplo, possuímos duas classes (e-mails e requisição de compra) que executam ações quando algo muda na classe de controle de estoque. É aí que entra a aplicação do **Observer**.

10.2 O padrão Observer

Definição

O objetivo desse padrão é a definição "um para muitos" entre objetos, na qual muitos objetos podem ser dependentes de um. Ele é composto por classes principais, chamadas de **Subjects** (sujeitos), e classes dependentes, chamadas de **Observers** (observadores).

Quando o objeto principal recebe alguma alteração em seu estado, todos os seus dependentes são notificados da sua atualização – tudo isso de forma automática. Entenda *alteração em seu estado* como a chamada de algum método ou alteração de alguma de suas propriedades internas.

O uso desde padrão reduz o acoplamento entre as classes, pois nenhuma das classes dependentes tem conhecimento das outras, como será mostrado no exemplo posterior. E o principal, a classe responsável por notificar os observadores também tem o menor acoplamento possível: tudo o que ela conhece é uma interface que todos os observadores compartilham. No final, isso acaba permitindo que todas as classes sejam usadas de forma totalmente independente.

O padrão é indicado quando mudanças em um objeto geram mudanças em outros (quantidade indefinida de outros objetos), mantendo um acoplamento baixo. Isso permite que todas as classes possam ser usadas de forma independente.

Diagrama de classes

A seguir, veja o diagrama de classes do padrão **Observer** baseado no nosso exemplo:

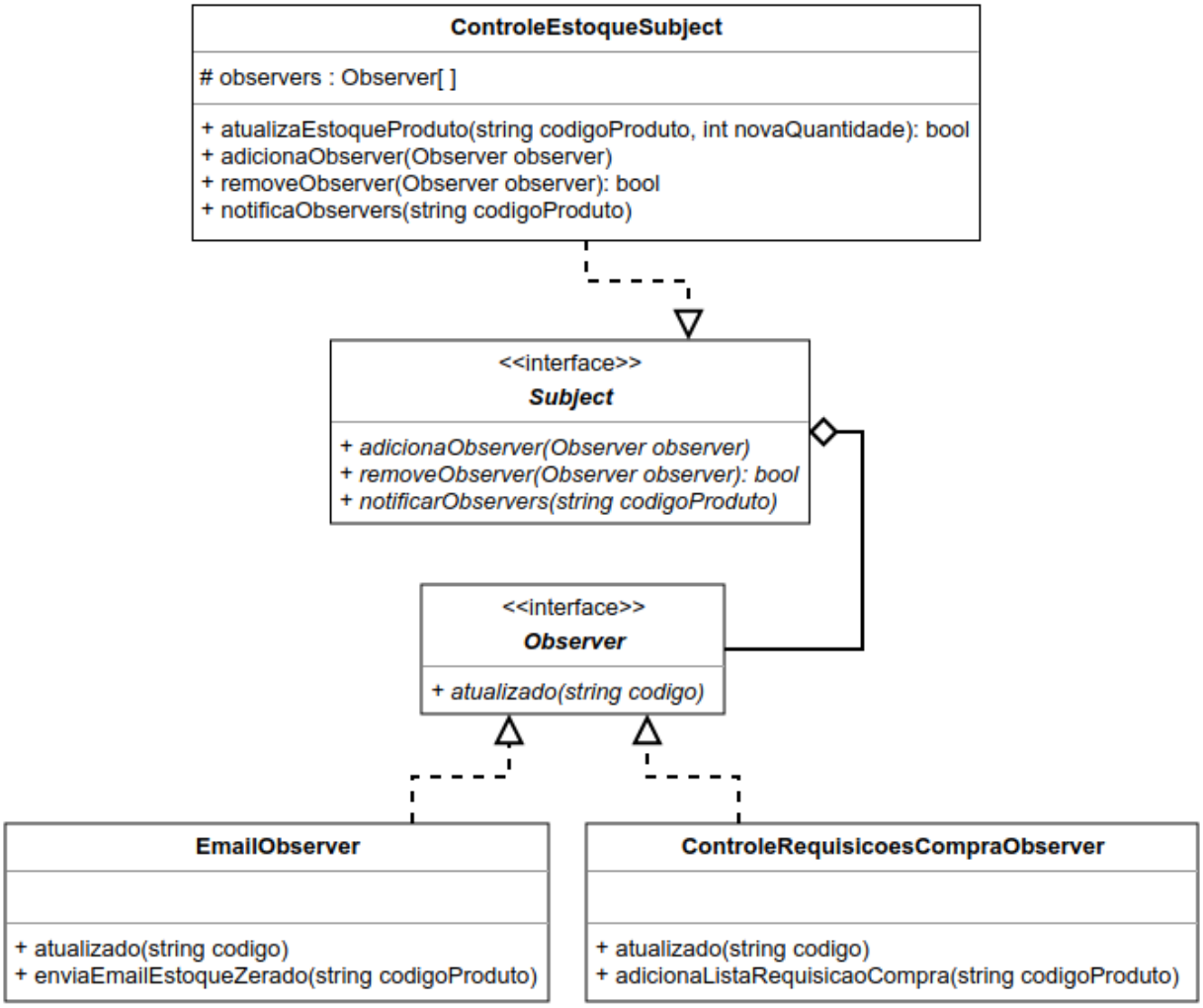

Figura 10.2: Diagrama de classes do Padrão de Projeto Observer

Elementos que compõem o Observer

- **Subject** (sujeito/assunto): interface que possui a definição dos métodos que devem ser seguidos por `Concrete Subject` ; é o contrato que garante a notificação da mudança dos estados dos sujeitos a seus observadores.
- **Observer** (observador): interface que possui a definição dos métodos de um observador concreto (`Concrete Observer`). Esse contrato garante que os observadores estarão no padrão para receber notificações dos sujeitos.

- **Concrete Subject** (sujeito/assunto concreto): classes concretas de sujeitos responsáveis por armazenar e enviar notificações a seus observadores.
- **Concrete Observer** (observador concreto): classes concretas de observadores que são responsáveis por receber notificações de sujeitos concretos (`Concrete Subjects`).

10.3 Resolvendo o problema da baixa de estoque

Vamos resolver a ideia do problema de atualização de estoque dos produtos do *ERP*. Criaremos uma classe que será responsável pela atualização do estoque, que é a necessidade principal da empresa, e depois criaremos as classes dependentes que serão os observadores da classe principal. Elas receberão uma notificação sempre que os estoques de um item chegarem a zero, tudo isso seguindo o padrão **Observer**.

Primeiro, precisamos criar o elemento `Observer` de nosso padrão, que será a interface que garantirá que nossos observadores estarão aptos a receber notificação dos sujeitos (`Subjects`), pelo contrato imposto pela interface.

Crie uma pasta `Observer` e, dentro dela, um arquivo com o nome `Observer.php` . A seguir, veja a implementação dessa interface:

```
<?php

namespace Observer;

interface Observer
{
    public function atualizado(string $codigo);
}
```

A interface `Observer` possui apenas a definição de um método, o `atualizado` , Ele sempre será chamado caso a classe principal

(sujeito) receba uma atualização em seu estado interno. O método não possui nenhum tipo de retorno especificado e recebe o parâmetro `$codigo` que é do tipo `string` e representa o código do produto com estoque zerado, que será usado pelos observadores concretos (**Concrete Observers**).

Uma observação muito importante aqui é que os parâmetros que serão passados para o método de notificação são definidos de acordo com cada necessidade. Não está implícito no padrão que esse método deve ou não receber algum tipo de parâmetro, pois isso não afeta a implementação. Definimos o código do produto como parâmetro, já que ele está diretamente ligado às ações dos observadores.

Agora que temos o nosso contrato a ser seguido pelos observadores, precisamos criar a real implementação deles (`Observers`) para que nosso padrão funcione. Crie um arquivo `EmailObserver.php` com o código a seguir:

```php
<?php

namespace Observer;

class EmailObserver implements Observer
{
    public function atualizado(string $codigo)
    {
        $this->enviaEmailEstoqueZerado($codigo);
    }

    public function enviaEmailEstoqueZerado(string $codigoProduto)
    {
        echo 'Email enviado para o setor de compras';
    }
}
```

A classe `EmailObserver` será responsável por enviar os e-mails para o setor de compras da empresa quando o estoque de um produto

chegar a zero. Ela representa o **Concrete Observer** de nosso padrão.

Note que ela implementa a interface `Observer`, e obrigatoriamente deve possuir um método com a mesma assinatura de `atualizado` dessa interface. Este método `atualizado` recebe o código identificador do produto e repassa para o `enviaEmailEstoqueZerado`, responsável por realmente enviar a mensagem de e-mail (em nosso exemplo, ele apenas simula o envio).

Vamos criar mais um exemplo de observador para seguir de acordo com o problema apresentado no início do capítulo, e para que possamos ver o sujeito trabalhando com mais de um observador (`Observer`). Crie um arquivo com o nome `ControleRequisicoesCompraObserver.php`, e siga o código:

```
<?php

namespace Observer;

class ControleRequisicoesCompraObserver implements Observer
{
    public function atualizado(string $codigo)
    {
        $this->adicionaListaRequisicaoCompra($codigo);
    }

    public function adicionaListaRequisicaoCompra(string $codigoProduto)
    {
        echo 'Produto adicionado à lista de requisições de compra';
    }
}
```

Assim como a classe anterior, `ControleRequisicoesCompraObserver` também é um observador concreto (**Concrete Observer**) e implementa a interface `Observer`. A diferença é que o método de notificação (`atualizado`) chama o método `adicionaListaRequisicaoCompra`, que também recebe um código identificador de produto e fica responsável por adicionar o produto a

uma lista de requisições de compra (em nosso exemplo, ele simula isso).

Agora que temos as classes do nosso observador (**Observer**) e sua interface pronta, precisamos construir a parte responsável por controlar os observadores e notificá-los a cada nova atualização. Primeiro, vamos criar a interface do sujeito (**Subject**). Esta será o contrato que garantirá que todos os sujeitos que a implementam poderão trabalhar com observadores que implementem a interface **Observer**.

Crie um arquivo com o nome de `Subject.php`, e siga o código:

```
<?php

namespace Observer;

interface Subject
{
`     public function adicionarObserver(Observer $observer);

    public function removerObserver(Observer $observer): bool;

    public function notificarObservers(string $codigoProduto);
}
```

A interface `Subject` possui três métodos. O primeiro é `adicionarObserver`, que recebe uma instância de `Observer`, ou seja, todas as classes concretas que implementarem essa interface vão funcionar como observadores. O objetivo desse método é adicionar um observador a um `array` que armazenará todos os observadores do sujeito.

O segundo é `removerObserver`, que também recebe uma instância de `Observer`, porém seu objetivo é remover o observador da listagem. O último método é `notificarObservers`, responsável por percorrer todos os observadores do sujeito e enviar notificações de atualização. Esse método recebe uma `string` com o código identificador do produto que será passado para os **Observers**, porém vale lembrar

que esse parâmetro pode variar de acordo com a necessidade de cada sistema – no nosso caso, precisávamos trabalhar com algo que identificasse o produto.

Agora que temos o sujeito (`Subject`), que é apenas um contrato, precisamos implementar a classe que de fato executará as ações relacionadas aos seus observadores. Crie um arquivo com o nome de `ControleEstoqueSubject.php` conforme a seguir:

```php
<?php

namespace Observer;

class ControleEstoqueSubject implements Subject
{
    private $observers;

    public function atualizaEstoqueProduto(string $codigoProduto, int $novaQuantidade): bool
    {
        // Simula a atualização de estoque de um produto.

        if ($novaQuantidade == 0) {
            $this->notificarObservers($codigoProduto);
        }
    }

    public function adicionarObserver(Observer $observer)
    {
        $this->observers[] = $observer;
    }

    public function removerObserver(Observer $observerRemover): bool
    {
        foreach ($this->observers as $index => $observer) {
            if ($observer === $observerRemover) {
                unset($this->observers[$index]);
                return true;
            }
        }
```

```
        return false;
    }

    public function notificarObservers(string $codigoProduto)
    {
        foreach ($this->observers as $observer) {
            $observer->atualizado($codigoProduto);
        }
    }
}
```

A classe `ControleEstoqueSubject` servirá para efetuar as atualizações de estoques dos produtos, e controlar e notificar seus observadores. Ela implementa a interface `Subject`, que a obriga a implementar os métodos `adicionarObserver`, `removerObserver` e `notificarObservers`.

Além dessas funções, criamos uma propriedade `$observers` como `private`, então nenhuma outra classe externa poderá alterá-la. Seu objetivo é armazenar sua lista de observadores. Se o PHP aceitasse definição de tipo em propriedades de classe, essa propriedade seria definida com o tipo de uma coleção de `Observers`. Isso garantiria que não fossem adicionados outros tipos de dados no local onde devem existir apenas instâncias que implementem `Observer`.

Porém, contornamos esse problema da não definição de tipos de propriedades com o método `adicionarObserver`, no qual centralizamos a adição de observadores. Não existe nenhum outro método que adicione observadores à lista além do `adicionarObserver`; ele recebe no parâmetro `$observer` uma instância de `Observer` e incrementa-o na lista de observadores pelo código `$this->observers[] = $observer;`.

Ao contrário do método anterior, temos o `removerObserver`, que, como o próprio nome já diz, é responsável por remover um observador do sujeito. Ele também recebe uma instância de uma classe que implemente `Observer`, depois percorre todos os observadores da lista; se achá-lo, então executa `unset($this->observers[$index])`.

O `unset` apaga a referência da lista, assim o **Observer** removido não será mais notificado. Por fim, o último método de nossa interface `Subject` é o `notificarObservers`.

Quando ele é chamado para notificar os observadores, ele percorre todos os observadores através do `foreach`, chamando a função `atualizado` de cada um e passando o código do produto que foi atualizado, conforme o código `$observer->atualizado($codigoProduto);`. Vale ressaltar que garantimos que sempre existirá a função `atualizado`, pois somente existirão objetos de classes que implementam a interface `Observer`, conforme já explicado.

Além dos métodos de nossa interface, temos o `atualizaEstoqueProduto`, que possui a regra de negócio de atualização de estoque dos produtos do *ERP*. O primeiro parâmetro, `$codigoProduto`, é do tipo `string` e representa o código do produto a ser atualizado. O segundo, `$novaQuantidade`, é um inteiro com a quantidade a ser atualizada do item.

Este método simula como se houvesse uma atualização real de estoque. Depois, existe uma validação `$novaQuantidade == 0`, que verifica se a quantidade a ser atualizada é igual a zero; caso seja, chamamos o método para notificação dos observadores com o código `$this->notificarObservers($codigoProduto);`, passando o código do produto atualizado.

A estrutura de arquivos ficou conforme a figura seguinte:

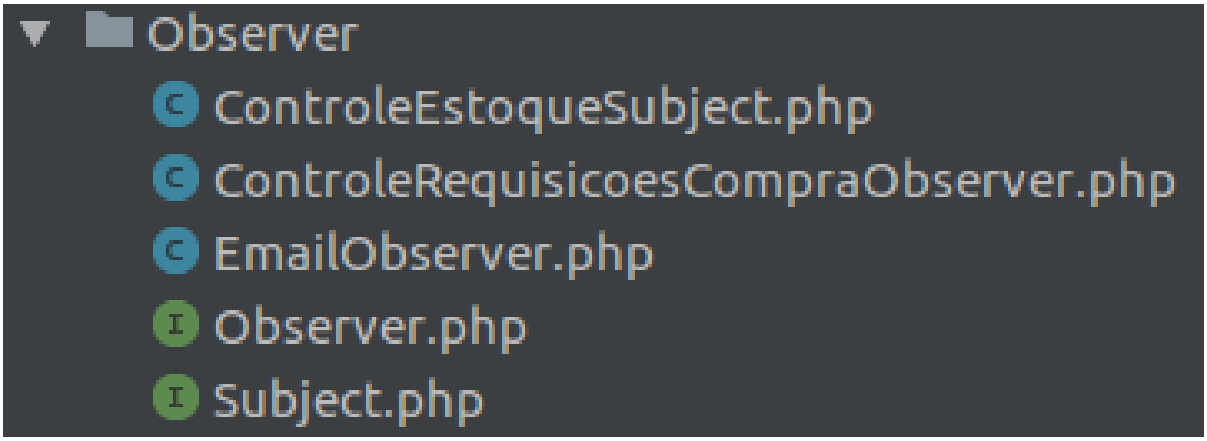

Figura 10.3: Estrutura de arquivos da implementação do Padrão de Projeto Observer

10.4 Conclusão

Em um sistema orientado a objetos, o padrão **Observer** torna-se perfeito quando possuímos um relacionamento "um para muitos" e precisamos executar ações em objetos distintos de acordo com modificações de um objeto principal. Essas ações são disparadas de acordo com uma parte de nosso sistema (sujeito observado) que sofre algum tipo de alteração.

Podemos até comparar o funcionamento desse disparo de notificações com um *broadcast*. Além de resolver essa necessidade, temos uma redução considerável no acoplamento entre as classes da nossa aplicação.

Capítulo 11
Variando partes de um algoritmo com Template Method

Tipo: padrão comportamental

Nível de dificuldade:

11.1 O problema do cálculo de impostos

Você está desenvolvendo um sistema de emissão de notas fiscais para venda de produtos. Primeiro, foram desenvolvidas as partes do sistema que permitem o cadastro de novos produtos e dos preços, o controle de estoque, entre muitas outras.

A única parte que falta desenvolver é a que calcula os valores finais do pedido e faz a emissão de notas fiscais. É aí que você começa a refletir sobre qual seria a melhor forma de estruturar essa parte do sistema.

A sua necessidade é que sejam selecionados produtos previamente cadastrados e suas respectivas quantidades. Após isso, o programa deve calcular alguns impostos fixos sobre a venda de cada um dos produtos e outros impostos que podem variar de acordo com alguns fatores. São dois impostos fixos (sempre são cobrados) e mais um que varia de acordo com o tipo de produto (vestuário, eletrônicos ou alimentação), sendo que cada um tem um cálculo (fórmula) específico.

Resumidamente, temos partes do algoritmo que são iguais para qualquer produto, e outras que variam, porém sempre vão existir. O padrão **Template Method** é perfeito para resolver este problema.

11.2 O padrão Template Method

Definição

O objetivo do Padrão de Projeto **Template Method** é definir uma base para um algoritmo que contenha partes que variam, sendo que essas variações reutilizam a base principal, que serve como uma espécie de esqueleto. A estrutura principal do algoritmo não se altera.

Temos uma classe abstrata com um método concreto chamado de "método template", dentro do qual existe a chamada a outros métodos da própria classe, porém são abstratos (que ainda não foram implementados). Ele também define a ordem em que os métodos serão executados.

Quanto à parte do nosso algoritmo que varia, ela fica implementada nas subclasses (que estendem a classe abstrata). Este código fica exatamente dentro dos métodos que foram definidos como abstratos na superclasse (classe abstrata estendida).

Uma das grandes vantagens de utilizar esse padrão é a reutilização de código, já que diminuímos consideravelmente a duplicação de partes do nosso algoritmo.

Hook Method (Método Gancho)

Como explicado anteriormente, temos uma classe abstrata que possui um método concreto chamado de "método template". Este faz a chamada a seus métodos abstratos, implementados nas subclasses (que as estendem).

Porém, algumas vezes temos a necessidade de executar alguma parte do algoritmo que será usada por apenas algumas dessas subclasses. Para que possamos chamar essa parte do algoritmo que será implementada em apenas algumas subclasses, temos os *Hook Methods* (Métodos Gancho).

O *Hook Method* é um método concreto definido na superclasse, que também é chamado diretamente do método template, mas geralmente sua implementação não faz nada; é apenas um método em branco. As subclasses que precisarem utilizar esse método vão sobrescrevê-lo. Desta forma, ele será chamado automaticamente no método `template`, conforme será mostrado no exemplo posterior deste capítulo.

Diagrama de classes

A seguir, veja o diagrama de classes do padrão **Template Method**:

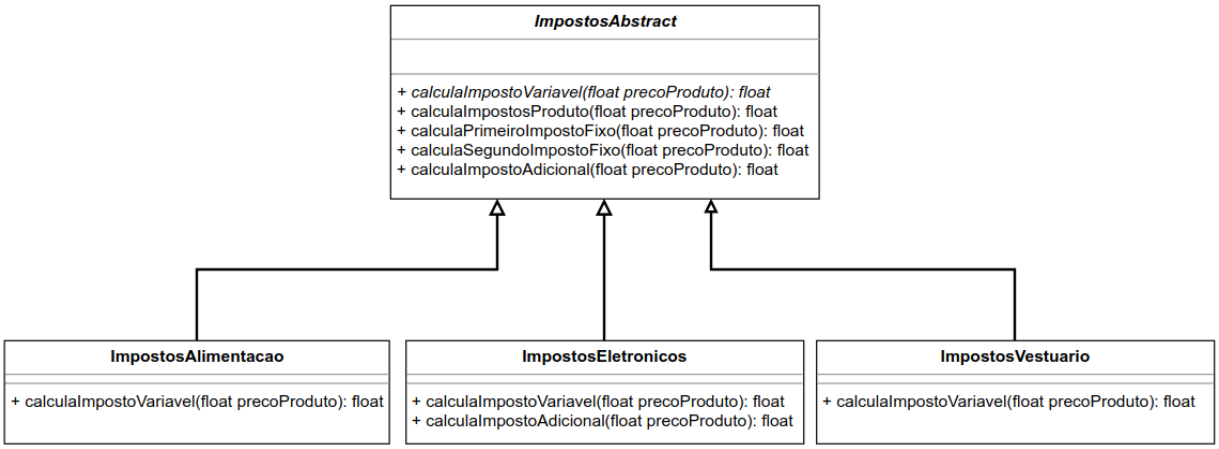

Figura 11.2: Diagrama de classes do padrão Template Method em PHP

Elementos que compõem o Template Method

- **Classe abstrata**: é o esqueleto do algoritmo, e contém a definição da parte principal do algoritmo que não varia.
- **Subclasses**: são as classes com a implementação do template que variam.

11.3 Resolvendo o problema do cálculo de impostos

Vamos implementar as classes necessárias para a resolução do problema do cálculo de impostos, apresentado no início deste capítulo, fazendo uso do *Template Method*. Conforme comentei antes, precisamos de uma estrutura de códigos que contenha a parte do código compartilhado entre todos os algoritmos.

Para começar, crie uma pasta com o nome de `TemplateMethod` e, dentro dela, crie um arquivo com o nome de `ImpostosAbstract.php`. Sua implementação deve estar conforme o exemplo a seguir:

```
<?php

namespace TemplateMethod;

abstract class ImpostosAbstract
{
    private function calculaPrimeiroImpostoFixo(float $precoProduto):
float
    {
        return ($precoProduto * 1.3);
    }

    private function calculaSegundoImpostoFixo(float $precoProduto): float
    {
        return ($precoProduto * 1.1);
    }

    abstract protected function calculaImpostoVariavel(float
$precoProduto): float;

    public function calculaImpostosProduto(float $precoProduto): float
    {
        $precoProduto = $this->calculaPrimeiroImpostoFixo($precoProduto);

        $precoProduto = $this->calculaSegundoImpostoFixo($precoProduto);
```

```
        $precoProduto = $this->calculaImpostoAdicional($precoProduto);

        $precoProduto = $this->calculaImpostoVariavel($precoProduto);

        return $precoProduto;
    }

    protected function calculaImpostoAdicional(float $precoProduto): float
    {
        return $precoProduto;
    }
}
```

Note que a classe `ImpostosAbstract` é do tipo `abstract`. Definimos desta forma porque precisávamos tanto criar o método abstrato `calculaImpostoVariavel`, que será implementado pelas subclasses, quanto implementar os demais métodos concretos. Com uma classe normal, não poderíamos criar o método abstrato e, com uma interface, não poderíamos implementar os métodos concretos.

Essa classe contém o método que representa o *Template Method* e, em nosso exemplo, ele é o `calculaImpostosProduto`. É nele que está definida a ordem lógica de execução das operações (cálculos dos impostos):

```
<?php

public function calculaImpostosProduto(float $precoProduto): float
{
    $precoProduto = $this->calculaPrimeiroImpostoFixo($precoProduto);

    $precoProduto = $this->calculaSegundoImpostoFixo($precoProduto);

    $precoProduto = $this->calculaImpostoAdicional($precoProduto);

    $precoProduto = $this->calculaImpostoVariavel($precoProduto);

    return $precoProduto;
}
```

Ele recebe como parâmetro um `float` com o preço do produto de adição de impostos. Na primeira linha, é chamado o método `calculaPrimeiroImpostoFixo`, que calcula e retorna o valor com um imposto em comum para todos os tipos de produtos.

Após isso, é chamado o método `calculaSegundoImpostoFixo`, que também faz o cálculo de um imposto em comum a todos os tipos de produtos. Também temos a chamada do nosso *Hook Method*, que é o `calculaImpostoAdicional`. Por padrão, esse método sempre retorna o próprio valor (sem alterações).

Na sequência, é chamado o método `calculaImpostoVariavel`, porém, se notarmos, não existe um método implementado com esse nome dentro da classe `ImpostosAbstract`, apenas um abstrato. É aí que entram as subclasses: elas serão as responsáveis pela implementação desses métodos; desta forma, cada uma poderá ter sua implementação (seu cálculo de impostos específico).

No final de todos os cálculos, o preço final com todos os impostos é retornado pelo código `return $precoProduto;`. O método `calculaImpostoVariavel` é definido como abstrato nesta classe, de modo que cada subclasse será obrigada a implementá-lo da sua maneira – contanto que ele receba como parâmetro e retorne valores do tipo `float`.

```
abstract protected function calculaImpostoVariavel(float $precoProduto):
float;
```

Para demonstrar a parte do nosso algoritmo, compartilhada por todas as implementações de subclasses, temos dois métodos:

```
private function calculaPrimeiroImpostoFixo(float $precoProduto): float
{
    return ($precoProduto * 1.3);
}

private function calculaSegundoImpostoFixo(float $precoProduto): float
{
```

```
    return ($precoProduto * 1.1);
}
```

Note que `calculaPrimeiroImpostoFixo` e `calculaSegundoImpostoFixo` são definidos com sua visibilidade como `private`, logo, ambos os métodos só poderão ser chamados de dentro da própria classe. O local que está chamando é o método template `calculaImpostosProduto`.

Não existe necessidade de duplicar sua chamada das subclasses nem de outros lugares de nosso sistema. Assim, o cálculo ficará centralizado nessa função, e evitamos a repetição de código.

Ambas as funções recebem um parâmetro do tipo `float`, que representa o valor do produto e retornam esse valor com o acréscimo dos impostos por meio de um pequeno cálculo matemático fictício.

Nosso último método desta classe a ser explicado é o `calculaImpostoAdicional`. Diferente dos anteriores, sua visibilidade foi definida como `protected`, pois ele poderá ser chamado de suas subclasses, já que essa é a ideia. Ele representa o *Hook Method*.

Note que ele é chamado dentro do método template `calculaImpostosProduto`, porém não faz absolutamente nada; ele apenas retorna o valor do produto que foi passado por parâmetro. Para as classes que quiserem adicionar algum tipo de imposto adicional, bastará sobrescrever esse método.

Ele será chamado no local dessa implementação da classe abstrata. Desta forma, ele será usado opcionalmente nas subclasses que necessitarem, como será mostrado posteriormente.

```
protected function calculaImpostoAdicional(float $precoProduto): float
{
    return $precoProduto - ($precoProduto * 0.1);
}
```

Agora, vamos construir o nosso primeiro exemplo de subclasse, que representará os cálculos de impostos para produtos de alimentação. Crie um arquivo com o nome de `ImpostosAlimentacao.php` . O exemplo a seguir demonstra a implementação da classe:

```
<?php

namespace TemplateMethod;

class ImpostosAlimentacao extends ImpostosAbstract
{
    protected function calculaImpostoVariavel(float $precoProduto): float
    {
        return $precoProduto;
    }
}
```

A classe `ImpostosAlimentacao` estende de `ImpostosAbstract` , e isso obriga-o a implementar o método `calculaImpostoVariavel` por ser a última classe da linha da herança (nenhuma classe abaixo).

Note que o método `calculaImpostoVariavel` apenas retorna o valor recebido por parâmetro. Colocamos desta forma porque queremos dizer que produtos alimentícios não têm impostos agregados. Caso a nossa instância da subclasse seja de `ImpostosAlimentacao` , então ela será a implementação a ser executada. Esta é a parte do nosso algoritmo que varia.

Vamos seguir para o próximo exemplo de nossas subclasses. Agora criaremos a implementação para o cálculo de impostos para produtos de vestuário. Crie um arquivo com o nome `ImpostosVestuario.php` e siga o exemplo:

```
<?php

namespace TemplateMethod;

class ImpostosVestuario extends ImpostosAbstract
{
```

```
    protected function calculaImpostoVariavel(float $precoProduto): float
    {
        return ($precoProduto * 1.15);
    }
}
```

Da mesma forma que a classe `ImpostosAlimentacao`, a `ImpostosVestuario` também deve estender de `ImpostosAbstract`. Entretanto, note que a implementação do método `calculaImpostoVariavel` está diferente: o exemplo anterior apenas retorna o preço do produto sem acréscimo de impostos, já este multiplica o preço do produto por `1.15`. Assim, se instanciarmos `ImpostosVestuario`, teremos o imposto adicional que apenas será agregado a produtos de vestuário.

Ainda não vimos nenhum exemplo que utiliza o *Hook Method*, então vamos implementar um terceiro exemplo de subclasse. Crie um arquivo com o nome `ImpostosEletricos.php` e siga o exemplo:

```
<?php

namespace TemplateMethod;

class ImpostosEletronicos extends ImpostosAbstract
{
    protected function calculaImpostoVariavel(float $precoProduto): float
    {
        $precoProduto = ($precoProduto * 1.2);

        return $precoProduto;
    }

    protected function calculaImpostoAdicional(float $precoProduto): float
    {
        return $precoProduto - ($precoProduto * 0.1);
    }
}
```

Da mesma forma que as classes anteriores, a `ImpostosEletronicos` tem sua própria implementação do método `calculaImpostoVariavel`,

que multiplica o preço dos produtos por `1.2`. Mas podemos notar que, nessa classe, existe um método a mais: o `calculaImpostoAdicional`, que representa nosso *Hook Method*.

Ele multiplica o preço do produto por `0.1`, e depois diminui esse resultado do preço do produto. Note que ele não é chamado nessa classe, porque já é chamado no método `template` da superclasse. A diferença é que os exemplos anteriores não haviam sobrescrito o comportamento desse método.

Para testarmos nosso código, crie um arquivo com o nome `index.php` e insira o código a seguir:

```
<?php

require_once('../autoloader.php');

$impostosAlimentacao = new \TemplateMethod\ImpostosAlimentacao();
$impostosVestuario = new \TemplateMethod\ImpostosVestuario();
$impostosEletronicos = new \TemplateMethod\ImpostosEletronicos();

echo "Preço final de alimentação: " . $impostosAlimentacao-
>calculaImpostosProduto(10);
echo "\n";

echo "Preço final de vestuário: " . $impostosVestuario-
>calculaImpostosProduto(10);
echo "\n";

echo "Preço final de eletrônicos: " . $impostosEletronicos-
>calculaImpostosProduto(10);
echo "\n";
```

Primeiro, estamos instanciando as três classes das nossas variações de algoritmos. As instâncias estão referenciadas nas variáveis `$impostosAlimentacao`, `$impostosVestuario` e `$impostosEletronicos`. Depois, chamamos o método `calculaImpostosProduto` de cada uma delas, passando o valor `10`, e damos um *print* na tela o resultado final.

O resultado esperado deve ser:

```
Preço final de alimentação: 14.3
Preço final de vestuário: 16.445
Preço final de eletrônicos: 15.444
```

Figura 11.3: Resultado esperado para o teste do Template Method

O fluxo que cada um dos algoritmos passou foi definido no método `template`, na seguinte ordem:

1. `calculaPrimeiroImpostoFixo` – Cálculo igual para todas as implementações;
2. `calculaSegundoImpostoFixo` – Cálculo igual para todas as implementações;
3. `calculaImpostoAdicional` – Só tem efeito nas classes que o sobrescreveram;
4. `calculaImpostoVariavel` – Cada subclasse tem sua implementação com as próprias regras.

A estrutura de arquivos ficou conforme a figura seguinte:

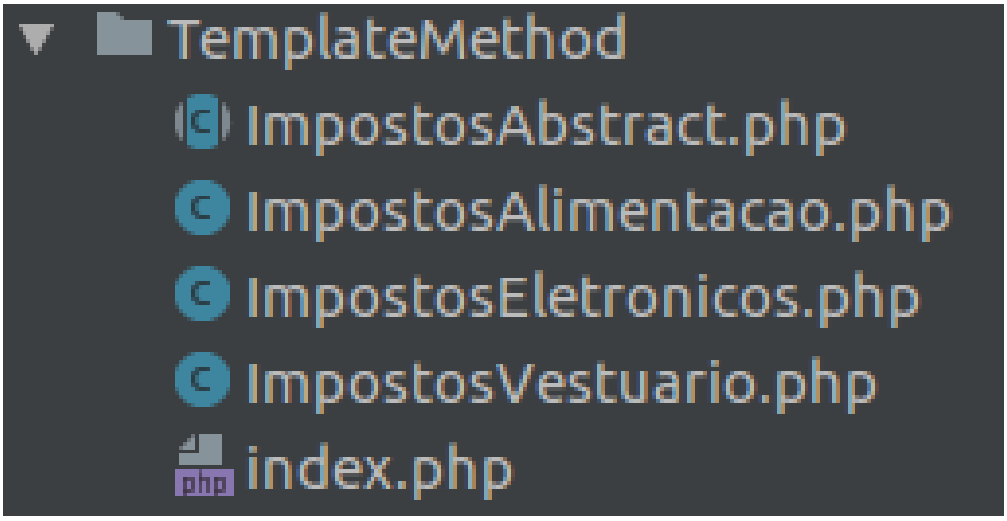

Figura 11.4: Estrutura de arquivos da implementação do Padrão de Projeto Template Method

11.4 Conclusão

Alguns pontos não tão positivos que podemos observar é que, às vezes, depurar e entender o fluxo da implementação do **Template Method** pode ser um pouco confuso. Desta forma, a manutenção pode ser um pouco crítica, e é possível uma pequena mudança nas subclasses afetar toda a estrutura.

Porém, apesar destes pequenos pontos negativos, o padrão **Template Method** torna-se uma excelente alternativa para o reaproveitamento de código, já que não há duplicidade das partes em comum entre nossos algoritmos. O padrão é construído com o uso da herança de classes, dando uma certa flexibilidade para que subclasses decidam como implementar partes específicas de um determinado algoritmo.

CAPÍTULO 12
A mudança de comportamentos em tempo de execução com Strategy

TIPO: padrão comportamental

Nível de dificuldade:

12.1 O problema da compressão de arquivos

Você trabalha em uma empresa que é contratada por terceiros para escanear documentos, uma prática muito comum em situações em que se quer digitalizar documentos físicos antigos. Quando uma nova demanda de documentos se inicia, existe um operador responsável por escanear manualmente folha a folha.

Estes arquivos são armazenados em um servidor da empresa e ficam organizados por pastas, e quem faz este controle é o próprio software da impressora. Neste mesmo servidor, existe uma aplicação desenvolvida em PHP rodando na web e, através dela, os clientes acessam um painel administrativo e têm acesso a todos os seus documentos.

Até então, os clientes sempre trabalhavam com uma quantidade não muito grande de documentos, e acessavam-nos diretamente online para a consulta, fazendo downloads um a um. Porém, recentemente um dos clientes solicitou a digitalização de milhares de documentos

e informou que seria necessário fazer o download de forma agrupada sempre que fosse necessário.

Uma solução pensada foi a de incluir um botão no painel administrativo que selecionasse todos os documentos e comprimisse em um único arquivo com a extensão `.RAR`, o que resolveria o problema. Após o analista do sistema refletir a solução, ele teve a ideia de aproveitar e já incluir a compressão dos arquivos no formato `.ZIP` e `.TAR`, assim, o cliente final poderia selecionar o formato que preferisse.

Pensando na estruturação desta parte do sistema, podemos notar que, independente do formato de saída dos arquivos, todos eles têm a finalidade em comum de gerar um arquivo comprimido. A única coisa que muda é uma parte do algoritmo, a parte que comprime para o formato diferente. Pensando em aproveitamento de código e em como poderíamos solucionar este problema de algoritmos semelhantes, que têm uma pequena parte do comportamento diferente, é que chegamos ao uso do **Strategy**.

12.2 O padrão Strategy

Definição

O Padrão de Projeto **Strategy**, também conhecido como **Policy**, tem como objetivo definir uma família de algoritmos que executam operações semelhantes (em comum), porém possuem diferenças quanto a sua implementação. Ele encapsula cada um desses algoritmos e permite que sejam utilizados por qualquer cliente, e que esses comportamentos sejam trocados em tempo de execução.

Muitas vezes, temos classes distintas, mas que diferem apenas em algum comportamento. Com o encapsulamento dos algoritmos (comportamentos), temos um maior reúso de código, pois extraímos

apenas a parte que varia, mantendo o restante em uma única classe que faz uso deles.

Os desenvolvedores costumam substituir o **Strategy** pela herança de classes ou uso de interfaces, já que o objetivo final dos comportamentos que variam e são reaproveitados é um tanto parecido, mas não tem exatamente o mesmo efeito. Seguindo o exemplo da herança, podemos ter classes com comportamentos distintos que estendem de uma classe em comum. Esta superclasse teria um comportamento concreto definido, e todas as subclasses utilizariam ou sobrescreveriam esse comportamento.

Herança de classes

A herança é um princípio utilizado em linguagens orientadas a objetos. Ela permite o compartilhamento de métodos e atributos entre classes distintas. Com ela, temos um maior aproveitamento de código, especialização de propriedades/atributos ou generalização de comportamentos.

Algumas linguagens permitem que uma classe estenda várias outras, e isso é conhecido como herança múltipla. O PHP não possui herança múltipla, porém, a partir da versão 5.4.0, é possível criar estruturas de código semelhantes a uma classe. Essa estrutura é chamada de **Trait** e foi criada para solucionar o problema da herança múltipla (http://php.net/manual/en/language.oop5.traits.php).

Agora, pense nas subclasses que não devem ter esse comportamento. Com a herança, elas obrigatoriamente teriam. Uma solução utilizada é sobrescrever o método que não se quer, e deixar sua implementação em branco, mas isso não é uma solução muito elegante.

A outra opção muito usada é a criação de uma interface com os métodos de cada comportamento. Ela é implementada apenas pelas

classes que necessitam daquele comportamento. Parece uma solução melhor que a herança, mas também seria um problema: imagine acabar chamando um método de alguma classe da mesma família de objetos e ele não existir, por não implementar a interface. Teríamos um erro em nossa aplicação.

É aí que entra o padrão **Strategy**. Ele faz uso da composição de objetos, assim, as nossas classes são compostas apenas de instâncias dos comportamentos dos quais necessita. Além dessa flexibilidade, temos a segurança de sempre existir um comportamento padrão a ser chamado e, com isso, evitar erros de execução em nossos programas. Tudo isso ficará mais claro nos exemplos a seguir.

Diagrama de classes

Veja o diagrama de classes do padrão **Strategy**:

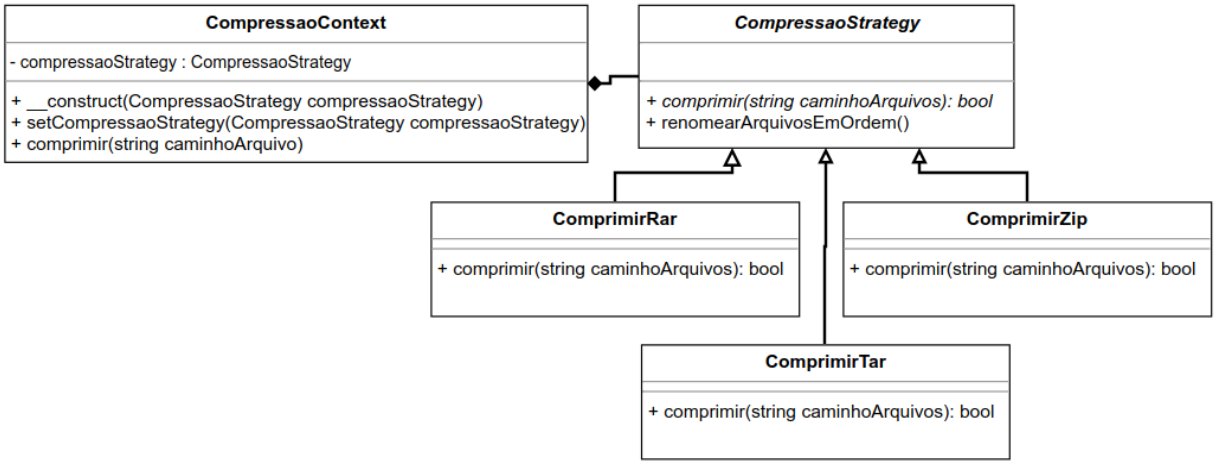

Figura 12.2: Diagrama de classes do Padrão de Projeto Strategy

Elementos que compõem o Strategy

- **Context** (contexto): é a classe responsável por criar e manter uma instância de nosso **Strategy**.
- **Strategy** (estratégia): serve de contrato para qualquer **Concrete Strategy** contendo os métodos obrigatórios

chamados por nossa classe de contexto. Pode ser tanto uma interface como uma classe abstrata. A classe abstrata é usada caso seja necessário implementar métodos em comum para nossas estratégias.
- **Concrete Strategy** (estratégia concreta): é a estratégia concreta que implementa a interface `Strategy`.

12.3 Resolvendo o problema da compressão de arquivos

Para montarmos a estrutura do **Strategy** e resolvermos o problema da compressão de arquivos, primeiro vamos separar os comportamentos que possuem uma finalidade semelhante. Em nosso exemplo, eles são os algoritmos responsáveis pela compressão de arquivos em diferentes formatos (`.ZIP` , `.RAR` , `.TAR`).

A finalidade semelhante é justamente a compressão dos arquivos, porém com implementação de código diferente. Esta é a análise inicial que devemos fazer antes de aplicar o **Strategy**, assim saberemos quando devemos utilizá-lo.

Primeiro, vamos definir o contrato que garantirá o padrão entre nossa família de algoritmos. Assim teremos a segurança de que cada algoritmo de compressão de arquivos possuirá um método que será chamado pelas classes que o usa.

Crie uma pasta com o nome de `Strategy` . Dentro dela, crie um arquivo com o nome de `CompressaoStrategy.php` e siga o exemplo do código a seguir:

```
<?php

namespace Strategy;

abstract class CompressaoStrategy
```

```
{
    public function renomearArquivosEmOrdem()
    {
        echo 'Arquivos renomeados!';
    }

    abstract public function comprimir(string $caminhoArquivos): bool;
}
```

Note que criamos uma classe abstrata, e que todas as classes com os algoritmos que representam nossas estratégias deverão estender dela. Porém, é importante deixar claro que o padrão também aceita uma interface no lugar da classe abstrata.

Devemos utilizar a abstrata quando temos um comportamento em comum em nossas estratégias. Neste caso, o método concreto `renomearArquivosEmOrdem` representa esse comportamento, e todas as subclasses herdarão esse método.

Em situações nas quais não possuímos um comportamento em comum, podemos simplesmente criar uma interface. Caso quiséssemos apenas gerar os arquivos sem antes ordená-los, não precisaríamos do método `renomearArquivosEmOrdem`. Logo, ter uma classe abstrata não faria muito sentido; teríamos apenas métodos abstratos.

No exemplo, temos o método `comprimir` definido como `abstract`, que recebe um valor do tipo `string` com o caminho dos arquivos a serem comprimidos e retorna um `boolean` de acordo com o sucesso ou a falha. Ele é o que terá a implementação de nossas estratégias, e todas as nossas subclasses implementarão de sua própria forma.

Mas o exemplo anterior é apenas a interface das estratégias. Agora precisamos das estratégias concretas, cada uma com seu comportamento. Vamos começar criando a responsável por gerar o arquivo de documentos no formato `.RAR`. Crie um arquivo com o nome de `ComprimirRar.php`, e siga o exemplo a seguir:

```php
<?php

namespace Strategy;

class ComprimirRar extends CompressaoStrategy
{
    public function comprimir(string $caminhoArquivos): bool
    {
        echo 'Arquivo comprimido no formato ".RAR"';

        return true;
    }
}
```

A classe `ComprimirRar` estende de `CompressaoStrategy`, o que faz dela a nossa primeira classe de estratégia. Note que temos apenas um método, o `comprimir`, que foi declarado como abstrato em nossa superclasse.

Geralmente, as classes de estratégia têm um método só, pois elas representam apenas o comportamento. Em nosso caso, o comportamento é o de comprimir os documentos em um arquivo compactado.

No nosso exemplo, estamos apenas simulando a compressão de um arquivo apenas para fins acadêmicos. Ao chamar esse método, será exibida a mensagem `Arquivo comprimido no formato ".RAR"`. Após isso, retornaremos o valor `true` simulando o sucesso na geração do arquivo. Em breve, entenderemos como isso será chamado.

Agora, crie um arquivo `ComprimirTar.php` e insira o código a seguir:

```php
<?php

namespace Strategy;

class ComprimirTar extends CompressaoStrategy
{
    public function comprimir(string $caminhoArquivos): bool
    {
```

```
        echo 'Arquivo comprimido no formato ".TAR"';

        return true;
    }
}
```

A classe `ComprimirTar` representa a estratégia de geração de arquivos no formato `.TAR`. Perceba que ela é muito semelhante ao exemplo anterior, porém seu comportamento é diferente: ao chamar o método `comprimir`, será exibida a mensagem `Arquivo comprimido no formato ".TAR"`.

Em um exemplo de código com uma funcionalidade real (além de um `echo`), poderíamos ter um código com muito mais diferenças entre ambas as classes de estratégia; poderiam ser estratégias de regras de cálculos extremamente complexas, por exemplo, mas tudo isso depende de cada necessidade.

Para finalizar o exemplo das estratégias, crie um arquivo com o nome de `ComprimirZip.php` e siga o exemplo a seguir:

```
<?php

namespace Strategy;

class ComprimirZip extends CompressaoStrategy
{
    public function comprimir(string $caminhoArquivos): bool
    {
        echo 'Arquivo comprimido no formato ".ZIP"';

        return true;
    }
}
```

A classe `ComprimirZip` também segue a mesma ideia das anteriores, mudando apenas o seu comportamento implementado no método `comprimir`. Ela exibe a mensagem `Arquivo comprimido no formato ".ZIP"`. Como estamos desenvolvendo apenas um exemplo, optei por apenas exibir a mensagem que simula a geração, supondo que

realmente houvesse a geração de um arquivo no formato esperado (`.ZIP`).

Além das estratégias, ainda falta implementarmos nossa classe de contexto, que ficará responsável por gerenciar e chamar nossas classes de estratégia. Crie um arquivo com o nome de `CompressaoContext.php` e siga o exemplo de implementação:

```
<?php

namespace Strategy;

class CompressaoContext
{
    private $compressaoStrategy;

    public function __construct(CompressaoStrategy $compressaoStrategy)
    {
        $this->compressaoStrategy = $compressaoStrategy;
    }

    public function setCompressaoStrategy(CompressaoStrategy
$compressaoStrategy)
    {
        $this->compressaoStrategy = $compressaoStrategy;
    }

    public function comprimir(string $caminhoArquivo)
    {
        $this->compressaoStrategy->comprimir($caminhoArquivo);
    }
}
```

Ao utilizar o padrão **Strategy**, sempre chamamos a classe de contexto em vez de chamar diretamente as classes de estratégia. A classe `CompressaoContext` possui o atributo privado `$compressaoStrategy` , dentro do qual será armazenada uma instância de `CompressaoStrategy` .

Desta forma, sempre teremos uma instância de nossas estratégias, e garantiremos que existirá um método `comprimir`, assim não vamos ter erros de execução no nosso código. Um ponto a ser observado é que estamos utilizando a composição de objetos. A estratégia está armazenada dentro da nossa classe de contexto.

O método construtor `__construct` recebe a instância de `CompressaoStrategy` (de suas subclasses), e lhe atribui a propriedade `$compressaoStrategy` para ser usada posteriormente. Quando passamos a instância pelo construtor, garantimos que sempre existirá uma estratégia na classe de contexto. Isso evita que o desenvolvedor crie uma instância sem nenhuma estratégia junto a ela, o que poderia gerar erros de execução.

Também implementamos o método `setCompressaoStrategy`, que é o *setter* responsável por substituir a instância da estratégia atual. Isso permite que modifiquemos a estratégia em tempo de execução. Poderíamos gerar um arquivo no formato `.ZIP` e, em seguida, apenas modificar a estratégia para gerar outro arquivo `.RAR`.

Para finalizar a explicação dessa classe, o método `comprimir` é responsável por chamar diretamente nossas estratégias. Ao chamá-lo, estamos chamando indiretamente uma classe de estratégia pelo código `$this->compressaoStrategy->comprimir($caminhoArquivo);`.

Como havia comentando anteriormente, pelo fato de a instância da propriedade `$compressaoStrategy` ser do tipo `CompressaoStrategy`, sempre existirá um método `comprimir`. Para testarmos o nosso padrão, vamos criar um arquivo com o nome de `index.php`, como o exemplo a seguir:

```
<?php

require_once('../autoloader.php');

$strategyRar = new \Strategy\ComprimirRar();
$strategyZip = new \Strategy\ComprimirZip();
$strategyTar = new \Strategy\ComprimirTar();
```

Criamos três variáveis com as estratégias `$strategyRar`, `$strategyZip` e `$strategyTar`, que serão passadas para a nossa classe de contexto. Agora, precisamos dessa classe responsável por manipular as nossas estratégias. Insira o código a seguir:

```
$compressaoContext = new \Strategy\CompressaoContext($strategyRar);
```

A variável `$compressaoContext` recebe instância de `CompressaoContext` e inicia a estratégia de geração dos arquivos de extensão `.RAR`.

```
$compressaoContext->comprimir('/CAMINHO/ARQUIVOS');
echo "\n";

$compressaoContext->setCompressaoStrategy($strategyZip);
$compressaoContext->comprimir('/CAMINHO/ARQUIVOS');
echo "\n";

$compressaoContext->setCompressaoStrategy($strategyTar);
$compressaoContext->comprimir('/CAMINHO/ARQUIVOS');
```

Chamamos o método `comprimir` de nosso contexto, que está com a estratégia de arquivos `.RAR`, e depois alteramos a estratégia para a geração de arquivos `.ZIP` com o método `setCompressaoStrategy`. Então, chamamos novamente `comprimir`, que está com um comportamento diferente.

No final, alteramos o comportamento para a geração de arquivos `.TAR`, e chamamos `comprimir` novamente para vermos a diferença. A saída esperada deve ser:

```
Arquivo comprimido no formato ".RAR"
Arquivo comprimido no formato ".ZIP"
Arquivo comprimido no formato ".TAR"
```

Figura 12.3: Resultado esperado para o teste do Strategy

A estrutura de arquivos ficou conforme a figura seguinte:

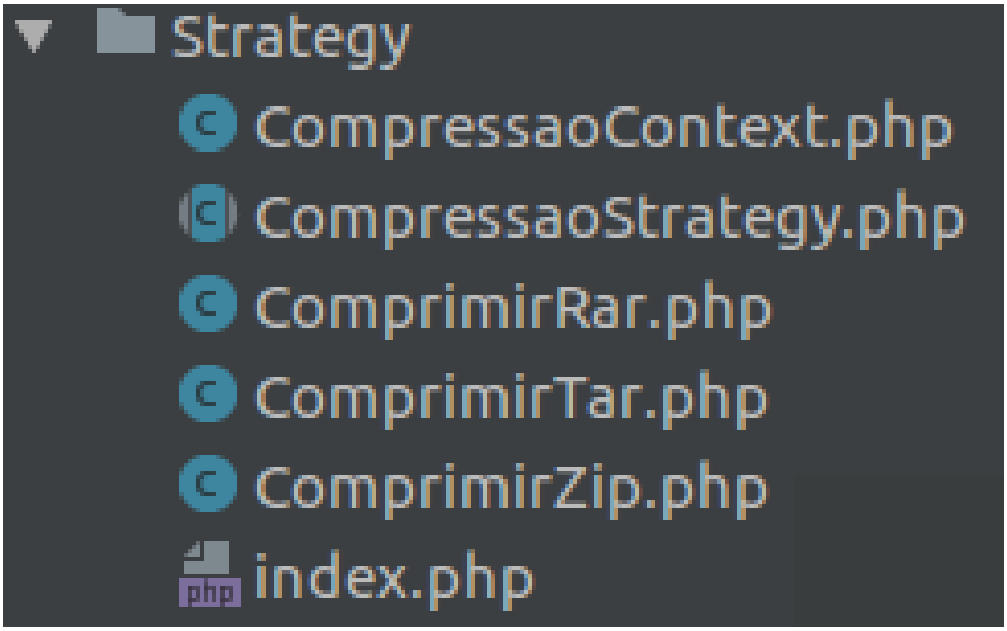

Figura 12.4: Estrutura de arquivos da implementação do Padrão de Projeto Strategy

12.4 Conclusão

O Padrão de Projeto **Strategy** é perfeito a ser implementado quando temos algoritmos semelhantes, que têm a mesma finalidade mas com comportamentos diferentes. Nestes casos, separamos o comportamento em classes específicas.

Também o aplicamos quando temos a necessidade de ocultar do usuário estruturas complexas de um algoritmo. Ele pode diminuir o uso de lógicas condicionais, e permite a variação dos algoritmos (comportamentos) em tempo de execução. Assim, esses comportamentos ficam mais desacoplados das classes que os executam, diminuindo a repetição de código e facilitando sua manutenção.

As desvantagens que podemos observar são o aumento no número de classes existentes no sistema e o aumento da complexidade de criação dos objetos. As classes de contexto têm como dependência

as de estratégia, então temos de instanciá-las e saber qual será usada. Em meu ponto de vista, as vantagens são muito superiores a essas pequenas desvantagens.

Conclusão

O mais importante de tudo não é decorar como cada padrão se aplica, mas sim qual é o melhor para resolver determinado problema. Sua implementação pode ser conferida neste e em outros livros, sempre que for necessário.

Capítulo 13
Dicas de leitura

A seguir, deixo algumas sugestões de leitura para que você aumente seu nível de conhecimento em alguns assuntos abordados neste livro.

Padrões de Projeto: soluções reutilizáveis de software orientado a objetos, ou *Design Patterns: elements of reusable object-oriented software* (em inglês).

Escrito por Erich Gamma, Richard Helm, Ralph Johnson e John Vlissides (famosa **GOF**), em 1994, é um dos livros mais conhecidos da engenharia de software, e a maior referência para livros que falam sobre Padrões de Projeto. O livro aborda todos os padrões descritos neste livro, entre alguns outros. Sem dúvida, é um item de coleção e manual de referência para se ter em sua cabeceira.

Use a cabeça: Padrões de Projetos, ou *Head First Design Patterns: a brain-friendly guide* (em inglês).

Escrito por Erick Freeman, Elisabeth Freeman e Kathy Sierra, em 2004, é mais um excelente livro que fala sobre Padrões de Projeto. Assim como todos os livros da série *Use a Cabeça*, este é um excelente livro com explicações detalhadas e exemplos um pouco fora da realidade, mas totalmente voltados para o aprendizado e para passar a ideia de qual problema cada padrão resolve. Os exemplos são escritos em Java.

Aprendendo Padrões de Projeto em PHP, ou *Learning PHP Design Patterns* (em inglês).

Escrito por W. Sanders, em 2013, é um excelente livro totalmente voltado para o PHP. Ele também traz alguns exemplos reais e esclarecedores. Foi desenvolvido quando o PHP estava em uma versão mais antiga e não chega a apresentar todos os padrões do **GOF**, porém é um ótimo complemento a este e todos os outros citados. Recomendo-o a todos desenvolvedores PHP que queiram aumentar sua visão sobre os padrões.

O codificador limpo, ou *The Clean Coder: a code of conduct for professional programmers* (em inglês).

Foi escrito por Robert C. Martin, em 2011. Em minha visão, todos que desenvolvem software profissionalmente, ou trabalham em áreas relacionadas, deveriam ler esse livro. Mais conhecido como o famoso *Uncle Bob*, o autor fala sobre o que é ser um profissional de software, boas práticas de desenvolvimento e diversas formas de melhorar como desenvolvedor. Confesso que, quando li a alguns anos atrás, mudei totalmente a minha visão em relação a alguns pontos.

Código limpo: habilidades práticas do Agile software, ou *Clean Code: a handbook of Agile software craftsmanship* (em inglês).

Também escrito por Robert C. Martin, em 2008, é um livro que foca em práticas de codificação para desenvolver um código limpo, falando de todos os benefícios e as práticas para isso. É mais um que recomendo ter em sua cabeceira, já que um código limpo e bem escrito só traz benefícios para os desenvolvedores que darão manutenções futuras, e para a sua empresa.

Capítulo 14
E agora?

14.1 Padrões do GOF não abordados neste livro

Além do conhecimento adquirido neste livro a respeito dos 11 **Padrões de Projeto** apresentados no decorrer dos capítulos, existem outros sugeridos pelo **GOF** no livro *Design Patterns* (GAMMA; HELM; JOHNSON; VLISSIDES, 1994). A seguir, estão listados esses padrões juntos da sua descrição resumida, retirada do livro original.

Chain of Responsibility – Padrão comportamental

Evita o acoplamento do remetente de uma solicitação ao seu destinatário, dando a mais de um objeto a chance de tratar a solicitação. Encadeia os objetos receptores e passa a solicitação ao longo da cadeia até que um objeto trate-a.

State – Padrão comportamental

Permite que um objeto altere seu comportamento quando seu estado interno muda. O objeto parecerá ter mudado sua classe.

Command – Padrão comportamental

Encapsula uma solicitação como um objeto, permitindo que você parametrize clientes com diferentes solicitações, enfileire ou registre (log) solicitações e suporte operações que podem ser desfeitas.

Interpreter – Padrão comportamental

Dada uma linguagem, define uma representação para a gramática juntamente com um interpretador, que usa a representação para interpretar sentenças nesta linguagem.

Iterator – Padrão comportamental

Fornece uma maneira de acessar sequencialmente os elementos de um objeto agregado, sem expor sua representação subjacente.

Mediator – Padrão comportamental

Define um objeto que encapsula como um conjunto de objetos interage. Esse padrão promove o acoplamento fraco ao evitar que os objetos se refiram explicitamente uns aos outros, permitindo que você varie suas interações independentemente.

Visitor – Padrão comportamental

Representa uma operação a ser executada sobre os elementos da estrutura de um objeto. Esse padrão permite que você defina uma nova operação, sem mudar as classes dos elementos sobre os quais opera.

Abstract Factory – Padrão de criação

Fornece uma interface para a criação de famílias de objetos relacionados ou dependentes, sem especificar suas classes concretas.

Bridge – Padrão estrutural

Separa uma abstração de sua implementação, de modo que as duas possam variar independentemente.

Composite – Padrão estrutural

Compõe objetos em estruturas de árvore para representar hierarquias do tipo partes-todo. Esse padrão permite que os clientes tratem objetos individuais e composições de objetos de maneira uniforme.

Flyweight – Padrão estrutural

Usa compartilhamento para suportar grandes quantidades de objetos, de granularidade fina, de maneira eficiente.

Proxy – Padrão estrutural

Fornece um objeto representante, ou um marcador de outro objeto, para controlar seu acesso.

14.2 Quando aplicar cada padrão?

Uma das maiores dificuldades relacionadas aos Padrões de Projeto é saber quando aplicar cada um deles, já que a aplicação de forma incorreta pode prejudicar toda a arquitetura de um sistema; em vez de facilitar, pode acabar dificultando sua manutenção.

Ao longo deste livro, procurei construir quase todos os exemplos da forma mais realista possível, com modelos de aplicações que já vi desenvolvidas em **PHP**, bem como algumas fictícias. Fato é que raramente teremos um exemplo igual em projetos com quais trabalhamos em nosso dia a dia, mas, no final das contas, muitas vezes teremos problemas semelhantes.

Cabe ao desenvolvedor conhecer qual é a proposta de resolução do problema que cada padrão visa solucionar. Não é necessário decorar cada detalhe de sua implementação, mas sim saber quando consultar este e outros livros que servem de referência.

Aplicação de Padrões de Projetos em frameworks

O primeiro ponto a ser observado é que todos os frameworks PHP mais populares da atualidade aplicam, ao menos, algum Padrão de Projeto em sua base. Entretanto, o foco aqui é discutir em relação à aplicação destas estruturas já prontas.

Ao optarmos por utilizar esses frameworks para construção de uma aplicação web, não precisamos nos preocupar com as funcionalidades mais básicas (padrão MVC, controle de rotas etc.), pois todas já vêm prontas para serem usadas. Porém, à medida que nosso projeto vai crescendo e vão surgindo necessidades de desenvolvimento, a aplicação de Padrões de Projeto pode se tornar uma excelente alternativa nestas novas necessidades.

14.3 Além da qualidade na aplicação de Padrões de Projeto

Caro leitor, é verdade que a aplicação de Padrões de Projeto é uma ótima prática a ser aplicada no desenvolvimento de um software. Aliás, quando combinada com algumas outras sugestões descritas neste livro e utilizadas pela comunidade de desenvolvimento, com certeza farão sua carreira decolar e o ajudarão a compreender mais a fundo o paradigma da Orientação a Objetos.

Mas, principalmente, os padrões o tornarão um desenvolvedor melhor, com um leque maior de conhecimento. Alguém que sabe as vantagens e desvantagens de cada uma das opções, que sabe como um framework realmente funciona, e não apenas saiba como utilizar suas funcionalidades, mas que conhece os problemas de um *software* e as melhores soluções disponíveis.

Peço que não pare sua busca por conhecimento por aqui. Leia outros livros relacionados da **Casa do Código** e busque diferentes autores com outras visões, para assim chegar ao ponto em que poderá tirar suas próprias conclusões.

Para finalizar, gostaria de dizer que acredito fortemente que são as suas decisões que determinam o seu destino, e não as condições – por mais que estas pareçam difíceis em algum momento. Quando comecei na carreira de desenvolvimento de software, ouvi no

primeiro mês que não era bom o bastante, e isso só me motivou mais ainda a seguir e superar as adversidades.

A força de vontade de correr atrás de seus objetivos e acreditar que é possível o levará a uma realidade muito melhor que a atual.

Capítulo 15
Referências bibliográficas

A seguir, estão listadas todas as fontes que utilizei como suporte para o desenvolvimento deste livro:

BRIZENO, M. *Padrões de Projeto.* 2013. Disponível em: https://brizeno.wordpress.com/padroes/.

FREEMAN, E.; ROBSON, E.; SIERRA, K. *Head First Design Patterns:* a brain-friendly guide. O'Reilly Media, 2004.

GAMMA, E.; HELM, R.; JOHNSON, R.; VLISSIDES, J. *Design Patterns:* elements of reusable object-oriented software. Addison-Wesley Professional, 1994.

MARTIN, R. C. *Clean Code:* a handbook of Agile software craftsmanship. Prentice Hall, 2008.

MARTIN, R. C. *The Clean Coder:* a code of conduct for professional programmers. Prentice Hall, 2011.

SANDERS, W. *Learning PHP Design Patterns.* O'Reilly Media, 2013.

SOURCE MAKING. *Design Patterns.* 2007. Disponível em: https://sourcemaking.com/design_patterns.